LDOS QUARTERLY

October 1, 1982 Volume 1, Number 6

In This Issue:

- The Communicating Micro
- The Lisp Language & LDOS
- Using Newscript 7.0 with Model 1 DDen
- Patching for LDOS Compatibility
- RTC by Roy Soltoff
- LDOS Disk I/O Routines

Table of Contents

INTRODUCTION FROM LSI:

EDITOR'S NOTES .... Page 2
VIEW FROM THE BOTTOM FLOOR .... Page 3
LSI NEWS .... Page 5

FROM OUR USERS:

EDAS IV for the Model I/III .... Page 9
The Communicating Micro .... Page 12
LDOS and BASIC file structure from a beginner's view .... Page 20
LISP Language and LDOS .... Page 26
Use JCL to control those FORTRAN and COBOL compilers .... Page 32
LDOS and the Hayes Smart Modem .... Page 34
Patching for LDOS compatibility - Visicalc .... Page 38
Using the LOOS disk I/O routines .... Page 43
LSCRIPT patches add versatility .... Page 45
Super-Short Terminal program + new ALIVE command .... Page 50
Using NEWSCRIPT 7.0 with Model I double density systems .... Page 55
PARITY = ODD - by Tim Daneliuk .... Page 59
.... er .... - Assembly Language Programming by Earle Robinson .... Page 64

FROM THE LDOS SUPPORT STAFF:

ITEMS OF GENERAL INTEREST .... Page 66
Includes TCHRON, TTIMER patch and Assembler code

RUN "PROGRAM",V - Chain those BASIC programs by Dick Konop .... Page 69

THE JCL CORNER - by Chuck .... Page 72
Patch LSCRIPT to be run with a JCL file

RTC - Roy talks about - - - .... Page 76
Information on many different system functions

LES INFORMATION - by Les Mikesell .... Page 83
Input from the RS-232 in LBASIC, plus a new *CL filter

LATE BREAKING NEWS, ETC .... Page 92

Includes SuperScripsit patch, FIX Disk news, and more

Copyright (C) 1982 by Logical Systems, Incorporated
11520 N. Port Washington Rd., Mequon, WI 53092
(414) 241-3066

EDITOR'S NOTES

This is the sixth and final issue of Volume One of the Quarterly. It marks the end of a year and a half of LDOS support and development. We have made many changes and improvements in both the Quarterly and the LDOS system in that time, and will continue to pass on new information in the pages of this publication.

Besides a new cover design, this issue of the Quarterly marks a major happening in its development - user contribution. Besides the regular user columns from Tim Daneliuk and Earle Robinson, we have a whole section of user contributed programs and articles. Maybe it was that offer of free software ....

Anyway, this issue contains many interesting articles, including a review of the LISP language and patches for Enhanced Visicalc. The normal staff columns are, as usual, full of many different things about the workings of LDOS. So, sit back and enjoy.

We are looking for reviews of any languages for our next issue. We currently have or will be getting reviews on LDOS "C", STSC APL, PASCAL 80, and ALCOR PASCAL. If you are currently using another language or one of the previously mentioned languages from a different author, we would be interested in seeing a review.

The LDOS Quarterly policy on the submission and payment for articles is as follows:

Articles sent for consideration should be accompanied by typewritten or lineprinted copy. An ASCII text file, Scripsit or SUPERScripsit file MUST accompany the printed copy! Please do not send in printed text without a disk, or send a file that has been created by routing the printer to a disk file!. No matter what the word processor used to create the file, it is much easier to format an original file than one that has been "printed" via a route of *PR. Payment will be made in the form of a product from LSI, or $25.00 per page ("page" is defined as page in the then current newsletter format). The size of the article will determine the value of the product, although no reasonable request will be refused. Please include your name, address, telephone number and LDOS serial number with your submission. LSI is extremely interested in seeing submissions from our users, and is open to suggestion on any ideas for the Quarterly.

Submissions should be sent to:

LDOS Quarterly Editor
11520 N. Port Washington Rd.
Mequon, WI 53092

The LDOS Quarterly is copyrighted in its entirety. No material contained herein may be duplicated for commercial purposes without the express written consent of Logical Systems, Inc. or the article's author.

V I E W   F R O M   T H E   B O T T O M   F L O O R

by Bill Schroeder

In my last column I asked for suggestions for possible new locations for Logical Systems. Many thanks to the people that responded with very interesting suggestions for that "perfect place". One person even suggested the North Pole..... for some reason I don't think he was a totally satisfied LDOS user..... oh well, we try. As of now, no decision has been made and will probably not be made until early 1983. We will keep you all informed as to our relocation plans.

As I am sure most of you know, the LDOS operating system (version 5.1.3) has been chosen by Radio Shack as the operating system that will be provided with their new Hard Drive system for the Mod I and III (cat. #26-1130). We are very proud that after seriously considering all of our competitor's products, support and companies behind those products, Radio Shack selected LDOS and Logical Systems. Also available through Radio Shack will be LDOS 5.1.3, both Mod I and III, floppy disk based systems, as catalog #26-2213 and 26-2214, respectively. This system will sell for $129 and will be virtually identical to the LDOS 5.1.3 available in the LSI packaging. LDOS and official LDOS support products from LSI will continue to be available through our worldwide dealer network. The only substantial difference between the LDOS sold by Radio Shack and the LDOS sold by our dealers is some statements in the manual and the absence of TWOSIDE/CMD on the Mod I system. The operating system itself is the same. Any program that is completely LDOS compatible will run just fine on the floppy and hard drive versions of LDOS provided through Radio Shack.

Many people have asked about how support will be handled for the LDOS product that is to be sold through Radio Shack. The answer is quite simple. Radio Shack will be providing support for LDOS as they do for all products they sell. But of course LSI will always be willing to assist any LEGITIMATE LDOS owner, no matter where that LDOS was purchased.

Many of our users have TRS-80s that are not quite TRS-80s such as PMC-80, PMC-81, Video Genie, LNW or Model III computers that have had NON-Radio Shack disk controllers installed. Unfortunately we do NOT have all these machines and are having a difficult time supporting these. The main problem is the foreign disk controllers in the Model III. These controllers, in most cases, do not function exactly the same as the official Radio Shack controller. The disk drivers in the LDOS system are designed and written for the TRS-80 and Radio Shack provided controllers. Use of LDOS with other controllers and/or computers may or may not work and may not always be as reliable as when used with the hardware that LDOS was written for and tested on. So please understand these problems.

We would like to help with any problem that an LDOS owner has. However, some problems are impossible for us to address, because we don't, in some cases, have the same hardware as the customer. We have only official Radio Shack hardware, all components (except disk drives) are pure Radio Shack. So when you consider new hardware please call or write. We will be happy to tell you if the product is supported by LSI or is known to work well with LDOS. If you don't check with us, you are on your own. When purchasing new software, make sure that the author or publisher will firmly state that the program in question IS IN FACT LDOS COMPATIBLE. If not, don't buy it and expect it to work with LDOS.

There are many LDOS users groups and special interest sub-groups popping up around the country. If you are involved with an LDOS group of any sort, please let us know where it is and how other users can get in touch with the group. In the next issue of the Quarterly we will be publishing a list of these groups. So let us know where they all are as soon as you can. Include: name of group, name of individual in charge, phone number, mailing address, meeting address, meeting times and dates and how to become a member.

In my last column, I mentioned our new LSI HOT-LINE (414/241-4100). We had some trouble with the answering machine that was to answer this line with a 3 minute message. We still don't have this 3 minute arrangement resolved but we have installed a machine with about a 1 minute message on that line. If you wish to call the HOT-LINE you will get this short message. We will have the longer message machine ready shortly, one way or another.

As of LDOS 5.1.3 there are no longer /FIX files included on the master disks. This is due to the changes needed in these fixes as the programs they are intended for change. There also would not be enough room on the disks for all the fixes we would like to provide. As a solution to this problem we will make available on October 1st, both hard copy versions and diskette versions of all available /FIX files. The hard copy set will be available for $5.00 and the diskette version will be $10.00. Postage will be paid by LSI on either version. (Foreign add $3.00 for Airmail). Either /FIX package will contain all of the official patches that have been developed to allow non-LSI products to function with LDOS. Some user contributed patches will also be included for products that have not been checked by LSI. It should be understood that this will be an ongoing process of patching programs and that the purchase of a /FIX package is strictly "AS IS" on the date purchased and no patch is guaranteed to produce any desired result. We will not offer updates or upgrades to these packages. If you require a patch that is developed after you purchase your /FIX package you will have to purchase the package again. With many thousands of users to service with these products, we must provide them in this way. Please check with us to confirm what patches are provided. To help you in determining if a particular patch is available we will publish a directory of the /FIX disk in each issue of this publication along with any new /FIX files, starting with the January issue.

Those who wish to have back issues of this publication can obtain them through LSI for $5.00 each postage paid. This is issue #6; we still have some copies of EVERY PREVIOUS ISSUE on hand for those who want them.

The FILTER concept is very important in the LDOS system and allows LDOS to do many things that cannot be done with other systems. Last year LSI offered a package called FILTER PACKAGE #1. It contained over a dozen useful filters WITH COMPLETE SOURCE CODE, at $60. So that even more of our users can enjoy the benefit of this package, LSI is permanently reducing the suggested retail price from $60 to $40. This super package has become even better with a new low price. LSI will provide you with complete info on this package for the asking. So don't forget the top notch FILTER package from LSI for just $40. Also check into FILTER PACKAGE #2 containing a whole new batch of handy filters for just $30. IMPORTANT: If you order both FILTER PACKAGE #1 and #2 at the same time you receive them both for just $60 (that was the old price of the FILTER #1 alone).

A NEW product that we are very proud of at LSI is The BASIC Answer (TBA). This is our BASIC program source processor which was a year in the making. BASIC programs can now be written WITHOUT LINE NUMBERS, use 14 CHARACTER VARIABLES, CONDITIONAL PROCESSING, GLOBAL AND LOCAL VARIABLES, and FULL CROSS REFERENCING during processor run and more. Don't miss out on this whole new world of BASIC programming. I'm so confident that this product will be the BASIC programmers best friend that I am offering a MONEY BACK GUARANTEE on TBA and a special TBA package. TBA has a retail price of just $69. During this special offer LSI will provide you with TBA and LED for just $79. That's right - buy TBA and LED (our text editor) together for just $79 (a $109 value). Don't delay too long though. This offer is only good for the rest of 1982. If you are not satisfied with this package for any reason, return it with the original invoice (postage paid) to LSI within 10 days of receipt and you will receive a FULL REFUND, no questions asked (But I would appreciate knowing what you did not like about TBA). This special money back guarantee is available on just TBA for the $69 price or the TBA & LED special at $79 until December 31, 1982. NOTE: This special offer is available only when purchasing directly from LSI. The first 50 TBA packages will also have a special FREE suprise included.

Radio Shack has now released the long awaited Super-Scripsit and Extended Visicalc products. I have spent some time using each and find the new Super Scripsit to be an excellent product worth much more than the asking price (note: some small bugs have been found, but should be corrected shortly). Both of these packages will run on LDOS with little change.

LSI is about to release several new products. Very shortly, we will be releasing TBA, FILTER PACKAGE #2, UTILITY PACKAGE #1, QUIZ-MASTER, THE LDOS USERS GUIDE, our new LDOS 513 REFERENCE MANUAL (Model I & III combined) and our /FIX products.

Many of our users have asked about our RAM BASED LDOS project and how it is coming. The answer is simple; it is ALIVE AND WELL. It will be using the LDOS SVC structure as is optional in 5.1, will reside below 3000H and will be called LDOS 6.0. No upgrades, exchanges, trade-ins or updates will be offered for this product. LDOS 6.0 should appear on several Z-80 ram based machines in the first half of 1983, and yes the TRS-80 Model II is being considered as one of the possible implementations. Arrangements are being worked out for offering a Microsoft compatible BASIC as well as an EDITOR ASSEMBLER, a "C" COMPILER, many system utilities and hopefully a PASCAL for use on LDOS 6.0. So standby - it is going to happen. By the way, this new LDOS will have FULL DEVICE INDEPENDENCE! You can even filter a file!!!

The popularity of the LDOS 5.1 system has grown to the point that there are books being written exclusively (or mainly) for the LDOS user audience. One of these is by Tim Daneliuk, a long standing LDOS user and a prominent author for BYTE, 80-MICRO, 80-US and INFOWORLD. His book will be called "LDOS - A Systems Guide" and should be available around the turn of the year. I have seen the first two chapters of this book and they look excellent. We will also see a work from a MAJOR industry author that will be published by IJG (Harv Pennington) that will be an encyclopedia of using LDOS & TRSDOS. This one will be a "biggy" so contact IJG in Upland, California if you are interested or want info on this one.

The Snapp TRIAL PACKAGE has gone over quite well. For just $10, you get complete documentation and a chance to try out ALL of the SNAPP modules. This is a very pleasant way to determine which, if any of these extensions to LBASIC will be of value to you. NOTE: this is a TRIAL package!! It will create only ONE functioning disk, good for a limited number of uses. When that disk wears out or fails, you will no longer be able to use the products and the $10 is not refundable. If you are interested in taking a look at the best BASIC extensions in the TRS-80 world (and their excellent documentation) contact Bob Snapp, toll free at 800/543-4628. As discussed in the "Late Breaking News" section at the end of this newsletter, the prices on all the SNAPP BASIC (LDOS type) products have been reduce by about 50%! The trial package and SNAPP Extended BASIC are available through LSI as well as through SNAPP Inc.

On September 23rd to 25th LSI hosted a very special TRS-80 industry meeting here in Mequon. A majority of the successful and innovative individuals providing products to the TRS-80 industry attended. The official participants were: Bob Snapp (SNAPP INC.), John Lancione (AEROCOMP), Kim Watt, Dennis Brent, Renato Reyes (POWERSDFT), Roy Soltoff (MISOSYS), Harv Pennington (I.J.G.), Irv Schmidt, Cameron Brown (80-US MAGAZINE), John Harding (MOLIMERX England), Tim Daneliuk (TRS-80 Author), John Dunn (Associated Services), Roger Billings, Kirk Hobart (LOBO DRIVES), Earle Robinson (SoftERware), Les Mikesell (MODEM-80), Bill Schroeder, Chuck Jensen, Dick Konop, Doug Kennedy, Rich Hilliard, Sue Dunn & staff (LOGICAL SYSTEMS). This meeting was very productive (exceptionally so for the non-Mequonites). 80-US magazine was on hand to cover this event and do interviews with the participants. These interviews will be published in upcoming issues of 80-US.

Speaking of magazines, I am often asked about the publications available to the TRS-80 user. I get all of these publications and find that the best all subject TRS-80 publication is, without a doubt, 80-US followed by TAS (The Alternate Source). Of the more general publications, EVERY Micro computer user should subscribe to INFOWORLD. For those wishing to get just one or two magazines my choice would be 80-US and INFOWORLD. For those of you who are not familiar with 80-US, we have arranged for you to get one FREE. All you have to do is write or call 80-US and give them your LDOS serial number. They will send you a FREE copy of 80-US for your review with NO STRINGS ATTACHED. They are at 3838 5. Warner St. Tacoma, Washington 98409 - (206) 475-2219. There is a coupon for this offer attached at the end of this newsletter. 80-US is also actively seeking articles directed towards the LDOS user. So if you are inclined to write and are an LDOS user, send your works to 80-US. I'm sure they will get prompt attention, and TIMELY PUBLICATION.

There is a company call Langley St. Clair Instrumentation that is providing both GREEN and ORANGE replacement video tubes for the TRS-80 Model I and III. We have both a green and orange tube installed here at LSI and are very pleased with them. We will be putting them in all of our TRS-80s this month. They are very easy on the eyes and are a snap to install (I'm not a tech and it only took me about 15 minutes total to put one in). These new tubes sell for as little as $79 and are worth every penny of it. For more info on these contact Langley St. Clair Instrumentation at 132 W. 24th Street, New York, NY 10011 - (212) 989-6876. Tell them Logical Systems sent you.

Last month our industry was shook by the passing of Harold Maush, the founder of PERCOM DATA in Dallas. Our sincere and heart felt sympathies go out to the Maush family and Harold's co-workers. The TRS-80 industry has lost a great supporter and innovator that will most certainly be missed by us all.

Thank you all for your on going support of LSI products. I hope that you and your families will have a pleasant and safe holiday season. Until next year...... (the January '83 issue)...... BYE.

LSI NEWS

This section contains a list of products available from LSI either now or in the near future. It also provides a timetable as to when a product will be available for shipment. Due to circumstances beyond our control, it is possible that products may not meet their stated delivery dates. LSI acknowledges this fact, but will make every effort to provide delivery on the date specified. Some of these products are provided by outside vendors over whom we have limited control. If a product is announced but is not available on the specified date, we hope you will have patience as we fully intend to provide all stated products/services eventually.

The following products have previously been announced and detailed in previous newsletters, and are either available now from LSI or will be available within the next 30 days:

```
The BASIC Answer (TBA) - $69.00  (see Page 4 for a special offer).
LDOS "C"               - $159.00
EDAS 4.x Mod I/III     - $100.00
Utility Disk #1        - $50.00
LDOS 5.1.3 Manual      - $59.00 (or $29.00 exchanged)
LDOS Operator's Guide  - $10.00
```

LDOS Operator's Guide

This is a short publication that is intended for the new LDOS user. It describes how to start using your TRS-80 with LDOS. In the future, it may be included with LDOS systems sold by LSI. Included are specific instructions on moving programs from other operating systems onto LDOS, creating working LDOS disks and making backup copies of them, and other information aimed at the novice user.

LDOS 5.1.3 Combined Manual

This is the LDOS manual that contains all the current changes in Library commands, Utilities, and the Technical section as relevant for version 5.1.3. The LBASIC section has been expanded to include descriptions and examples of all disk BASIC commands as well as the complete disk BASIC error dictionary listing. The JCL section has been re-written and grouped into beginning and advanced sections, with certain areas explained in greater detail. The Technical section has been restructured to include all Model I and Model III information, including a listing of both machines' entry point addresses and SVC's (where applicable) and all new system vectors and entry points. The layout is also in a more logical order. For those of you doing development on both machines, it is definitely a worthwhile investment at $29.00 + $4.00 S&H, exchanged. To order this manual, you should send in the proper amount along with your old manual. Be sure to KEEP your binder, tab inserts, and addendum section, as these will not be provided with the new manual.

UTILITY DISK #1

This disk will contain a multitude of useful programs, some of which have been requested by our users. As of press time, the contents of the disk are as follows:

COMP/CMD will be a file/diskette compare program. In the file compare mode, it will do a sector by sector compare of two files and send the differences to either the video or printer. It will also compare two diskettes sector by sector.

DCT/CMD is a utility primarily useful for those developing disk drivers or using non-standard disk drives. It provides an easy method to set up a DCT entry.

DIRCHECK/CMD will check a disk's directory, looking for incorrect GAT and HIT entries, as well as checking for bad file FPDE/FXDE chaining. Certain types of errors may also be corrected.

FIXGAT/CMD will re-create an unusable directory GAT sector. The user will be prompted for certain information, such as the number of cylinders, density, and number of sides on the diskette.

TYPEIN/CMD is a combination of JCL and KSM. It allows the user to have a specified string of characters or commands fed into the system to control operation. Unlike JCL, the characters can be fed to any program requesting keyboard input. This means that even Visicalc, Lscript, LBASIC INKEY$ statements, etc. can be answered. The characters can be directly typed in from the keyboard or can be picked up from a previously constructed file. It even can be called in the middle of JCL file, and can convert a specified number of lines to single character input.

HIGH/CMD will show a display of high memory usage, as long as the modules conform to the standard LDOS memory header format.

MAKEFILE/CMD is similar to the CREATE Library command in that it allows allocating space for a file by specified size in K or by the number of records of a given LRL. It also will allow file to be filled with a specified byte. The file can be marked as wither CREATEd or normal.

MAP/CMD will display a list showing where on the disk a file is stored. It will be broken down by extents and sector used in each extent. Killed files may also be mapped, including an indication if any of their previously used allocation is currently in use by some other file.

RAMTEST/CMD is a memory test utility that tests both high memory and that used by the resident LDOS system.

READ4080/CMD will allow a 40 track disk to be read in an 80 track drive. This will allow copying files from a 40 track disk with the COPY command and the BACKUP and CONV utilities.

READII will display a directory of and copy files from a Model II TRSDOS disk.

RDTEST/CMD will do a non-destructive read test of a diskette, providing a verification of all tracks and sectors.

RWTEST/CMD will do a write/verify test of a diskette. This will be useful for checking the operation of a disk drive or diskette media.

UNKILL/CMD will restore a file that has been accidentally KILLed or PURGEd.

LC - The LDOS "C" Language Compiler

LC is an integer-only implementation of C which provides all C statements except "struct", "union", "goto", "switch-case", and "typedef". All data types except "float" and "double" are implemented; "long" and "short" declarations are accepted, but 16-bit fields are used for all integers. In LC, "char" variables are implicitly unsigned. Single-precision and double-precision floating point operations are supported via functions supplied in the FP/LIB library included with the LC compiler. LC accepts multiple input files, with four levels of nesting for "#include'd" files. The compiler generates an EDAS Version IV assembler source file which is then assembled with the standard library and any other libraries needed to resolve function references in order to generate the executable program. The value in generating assembler source is twofold. First, you can obtain a complete machine code source listing which could prove invaluable in debugging complex code. Second, local optimization of assembler source code can be performed as required by the experienced assembler programmer. The LC standard library provides such functions as standard I/O redirection, dynamic memory allocation, automatic standard I/O opening and closing, and program chaining. In addition, functions specific to LDOS and the Model I/III are supplied in an installation library, to provide access to such functions as graphics and system entry points.

LC supports separate compilation; programs may be compiled in segments, and frequently used functions can be pre-compiled. You can create your own library of commonly used functions with the Partitioned Data Set utility (PDS is not included with LC but is available as a separate package). The assembler source code output by LC is designed to use the extensive SEARCH and conditional assembly support in EDAS Version IV. The assembler and companion assembler cross-reference utility are supplied with the LC package. You need nothing more to start writing and running C-language programs except your LDOS-equipped computer and a copy of the book "THE C PROGRAMMING LANGUAGE" by Kernighan and Ritchie. A 48K-RAM two-drive Model I (lower case video) or Model III is required.

Some highlights of the "elsie" compiler are:

- Integer subset of the C language.
- Access to floating point routines in ROM via function calls.
- All statements supported except STRUCT, UNION, TYPEDEF, SWITCH-CASE, GOTO.
- All operators supported except "->", ".", SIZEOF, and (TYPENAME).
- UNIX-compatible standard I/O library.

- o Standard I/O redirection with complete device independence.
- o Input using FGETS or GETS functions support LDOS Job Control Language.
- o Dynamic memory management (ALLOC, FREE, SBRK).
- o Sequential files open for READ, WRITE, and APPEND.
- o Generates Z-80 EDAS Version IV source code as output.
- o User libraries in Z-80 source ISAM-accessed PDS files.
- o Compact one-line invocation of the compiler.
- o LC's interactive friendly interface provides easy way to learn LC options.
- o Supports separate compilation of functions.
- o Compiled programs run under both Models I and III without modification.
- o Installation library gives access to graphics and LDOS entry points.
- o Supplied with example programs and utilities in source form.
- o LC/LIB includes: FPRINTF, PRINTF, ALLOC, FREE, SBRK, and String functions.
- o The LC package is Model I/III LDOS compatible and includes LC/CMD, LC/LIB, FP/LIB, IN/LIB, EDAS-IV, XREF, and more than 200 pages of documentation.

A New Version - EDAS-IV

by Marc Leager

The following is a review of some of the features of the new Model I/III EDAS 4.1.

Some programs are terrific when they are first released, then a new version comes out which is five times better than the original. LDOS is one of these programs. EDAS is another. MISOSYS has recently released a new version of its editor/assembler called EDAS-IV. It is the most exciting piece of software I have seen in a long time. It not only provides the best editor/assembler around, but combines many of the latest features of micro-computer program storage into a single working system. Please read on to discover what the new EDAS can do for you.

First we need an introduction for those who have not yet used EDAS. As its name implies, EDAS is a program editor and an assembler. The primary use for EDAS is in maintaining assembly language programs. But wait! As you read further you will see how the new EDAS can be used for other languages also. The traditional EDAS is a superior program editor for assembler language programs. It loads the entire program in memory so there are no disk accesses during an edit session for a single module. Line numbers are maintained for all program lines, so reference to a printed listing of the program during the edit session is possible. Best of all, EDAS allows all input to be in lower case. It automatically converts the actual language instructions to upper case, leaving comments and strings in their original typed form. At any time, the module in memory can be assembled by pressing a single key. If the module is a complete program, then the assembly is done instantaneously without disk access. Some programs are large and must be stored in several modules. For these, EDAS is able to load modules into the work space using the *GET operator to assemble the full program.

The first reason for using EDAS is that it works well. Other editors for assembly programs constantly turn the disks on for reading another 256 bytes of source, and tend to scramble the code when modules get larger than 0 code lines. EDAS is quiet and doesn't wipe out your hard fought code lines. Another reason is its speed. Assembly with EDAS is very fast for single-module programs. Even for multi-module programs, it is faster than assembling modules separately, then linking them together with a linkage edit program. A third reason for using EDAS is its friendliness. Mentioned before was the upper/lower case support. This produces a very readable program where code is upper case and comments are lower case. The search, insert, edit, print and display commands are also easy to use. They work quickly and correctly every time.

If you are wondering whether you would get any use out of this sophisticated editor/assembler, consider these items. LDOS is a fully documented, wide-open operating system. We are encouraged to write our own filters and drivers and special routines. With LDOS you can easily open and read DIR/SYS from any language, and from

assembly language you can read or write ANY part of the disk if you want to. Also using assembly language, you can write filters, drivers, and special routines to fit your individual needs. LDOS is designed for this type of user enhancement. (Note: LC, the "C" compiler, will include EDAS-IV as its assembler. "C" is another language which is well suited for system-type programs.) If you have been thinking of taking full advantage of the LDOS architecture by writing a few assembly programs for special effects, EDAS is a good editor/assembler for this purpose.

Now for descriptions of some of the outstanding features of EDAS-IV. I assume you have a moderate understanding of the basic purpose of the original EDAS. Some of the features described here are extensions of the original, and some are new. Since the most exciting features are entirely new, this article should still be informative for many of its readers without prior EDAS understanding. First is the treatment of line numbers. EDAS-IV supports line numbers similarly to EDAS 3.5.2, but the default for EDAS-IV is unnumbered lines. This may be startling to some, but now EDAS-IV can read any valid source file, numbered or unnumbered, without any specific action by the operator. Previously, you had to know whether each source file had line numbers, and also whether it had the standard EDAS 7-byte header. Now all of this is handled automatically! The reason for preferring unnumbered lines on disk is to save file space. In a typical source file, the line numbers take up 16% of the space. It also allows EDAS-IV to process the source for any language which does not require line numbers in the source file. One more wrinkle with line numbers is that if you want the numbers, and you intend to use the M-80 assembler, EDAS-IV has an option for using the M-80 line number format when the file is saved. This is imperative when you are using M-80 to produce listings of your assembled programs.

Some new commands have been added with EDAS-IV. First is Copy. You may wonder how this is done since the <C> operator is already in use for string replacement. The answer is the <C> now does both functions. If you follow the <C> with a number, then EDAS-IV assumes you are entering line numbers for a copy operation. Otherwise, it processes your keym as a string replacement operation. Due to popular request, the <Z> operator has been added. You may wonder what <Z> does, and the answer is that it does anything you want it to. EDAS-IV has a 50-byte patch area just for users, and <Z> can be used to jump to this patch area. I have been thinking of using this function for several things, such as showing the name of the module currently in memory, or changing the default extension during an edit session, or presetting assembly options, or plugging in my alternate keyboard routine, or ... Another keyboard feature in EDAS-IV concerns the BREAK key. In EDAS 3.5.2, if BREAK was the last key pushed before a Write operation, the BREAK could sometimes reappear during the write, aborting the write and producing a null file on disk. This was very disturbing, since there was no indication of this event to the operator. Now in EDAS-IV, all is well with BREAK correctly handled.

Now we will begin some new features in EDAS-IV. The *GET operator has been enhanced and a *SEARCH operator has been developed. These operators tell EDAS-IV during an assembly to read more source code from files on disk. They are a way of expanding the module in memory to larger than memory size, and of including standard functions into an application program. The *GET operator just reads the named source file and inserts it in the current module at the current location. The enhancement to *GET is to allow nesting of up to five levels. This allows you to collect subroutines or macros (!) from separate locations into standard sets using a series of *GETS. The sets can be included en masse by naming the desired collection with a single *GET in your source module. *GET can appear at any point in your program; in the beginning for declaring equate names or defining macros, or in the body for loading standard subroutines. The *SEARCH operator is similar to *GET since it accesses files stored on disk, but what a difference! *SEARCH doesn't read sequential files. Instead it processes members from a partitioned data set, PDS. This is an exciting feature since *SEARCH will only load the members from the PDS which are referenced in your program. This means you can collect your standard subroutines into a PDS library, then search the library when assembling any program. Advantages are that the PDS saves space on disk, allows standardization of your subroutines, encourages centralization into a small number of subroutine libraries, and makes your application source much cleaner. You can even

begin packaging your subroutine libraries for exchange or distribution among other EDAS-IV users, for mutual benefit. The net effect of the *GET and *SEARCH operators is to standardize and reduce the effort of writing new code.

The last new feature we will cover here is Macros. Of all the enhancements in EDAS-IV, this is the most exciting. You may wonder why it is mentioned last. The reason is that macros are a very complex feature, and I felt that some groundwork was needed before introducing them. Despite the complexity, the implementation in EDAS-IV is comprehensive; probably the best you will find in any micro-computer assembler. First, we will define a macro as an expression in a single line of code that is expanded by the assembler to produce many lines of code. The LDOS JCL feature is somewhat similar to assembler macros. Using a macro, you name none, or one or more values which are to be assigned to variables in the macro. The assembler then takes the values, inserts them in the necessary lines of code contained in the macro, and produces useable source code for further assembly. Doing this, you can tailor the effect of any macro by changing the values you use for parameters. For instance, a macro to move data from one area to another might be

```
MOVE    MACRO   #FR,#TO,#LG
        LD      HL,#FR
        LD      DE,#TO
        LD      BC,#LG
        LDIR
        ENDM
```

To use this macro you could code ... MOVE FRAREA,(TOPTR),50 This single statement would be expanded to the four Z-80 instructions needed to perform the move. Notice that one area was defined by name, one was defined by a pointer, and the length by an explicit value. This shows the way you can use a standard macro to produce code from a wide variety of value expressions. The macro processor in EDAS-IV covers all bases in code expansion by allowing both positional and keyword notation. This means it is very flexible in the structures it can handle. For example, the above macro call could have been coded ... MOVE #FR=FRAREA,#LG=50,#TO=(TOPTR). These features alone are extremely useful to the assembly programmer, but EDAS-IV does more. It has implemented a full range of tests and comparisons on the macro variables for conditional logic within the macro expansion. With this you are truly able to generate any code structure desired. For instance, you can test how many parameters were coded in the macro statement, you can test the value of any parameter against a string or against an assembled label. You can test to see if a label named by a parameter is defined elsewhere in the program, or is referenced elsewhere. The tests can be nested to eight levels. The ELSE operator is available for either/or constructions. I may have left out some features, but EDAS-IV seems to have enough strength to handle the most sophisticated macro construction imaginable. Some macros I have already written do the following: initialize a program to run on the Model I or III, move data (see above), create a BASIC dummy program so an assembly program can define arrays in BASIC, read a line from a disk file similar to the @KEYIN vector for the keyboard, and on and on.

At last, a summary. You should be very excited by EDAS-IV for several reasons. The macro processor is super. You can plan to use it for simplifying your coding. As in the MOVE example, it reduced the code lines by 75 percent. You can build a macro library, again using EDAS, then access it with *GET for each assembly. By putting *LIST OFF at the head of the macro library, and *LIST ON at the end, you can avoid having the text of the macros print in your program listings. The *SEARCH feature is also a terrific addition. You can build one or more subroutine libraries, then let EDAS select subroutines only as needed from them. (Note that the MISOSYS PDS program is required if you plan to build your own libraries.) These two features, macros and *SEARCH, should make assembly language much easier for many of us. Unfortunately, there is even more in EDAS-IV that could not be included in this review. You can be sure that the other features not mentioned are just as well implemented as the superstars we covered here. In closing, let me say that I am really enjoying EDAS-IV and hope that you will too. Maybe we can share an idea or a subroutine or a macro sometime as the new EDAS makes us more proficient programmers.

THE COMMUNICATING MICRO

Gordon B. Thompson,
Bell-Northern Research, Ottawa, Canada

Even amongst the experts, a communicating personal computer is perceived as being a small computer that has been programmed to behave as a smart terminal for accessing large mainframe computers, or other similar services, via a communications link. The terminal programs that are on the market today stress this "mother and child" relationship, although most do offer some kind of symmetrical facility for transferring files.

As small personal systems proliferate, other patterns of intercommunication can begin to emerge. However, if the models that the users have of the basic communication processes are too simplistic, they will fail to see the utility of those richer modes of interaction.

The Shared Space Model of Communications.

The common, familiar telephone creates a common acoustic space that can be shared by two people, who may be many miles apart. The calling and the called parties can interrupt, talk simultaneously, and even share a cry, or whatever. As evidenced by their behavior, this simple property of the familiar telephone, this sharing of a common acoustic space, is not conceptually understood by most users.

The shared space model of interpersonnal communication recognizes the prime importance of periods during which both communicants are talking, overlapping each other. The simple two wire telephone allows both parties to talk simultaneously, should they so desire. When this happens, we can say both parties are sharing the common acoustic space that the telephone provides. They are occupying the same space, at the same time.

Perhaps one of the reasons Picturephone failed to achieve any significant market acceptance is that it really added very little to the size of the common information space that was shared between the communicating parties. That the Picturephone doesn't have a shared visual space can be demonstrated by considering what must be done to play a simple game of tic-tac-toe. When we start, you have the grid which I can see on my screen. Where do I put my cross? On the screen? That won't work because you would not see it. No, I must copy your grid onto a sheet of paper, viewed by my camera, and place my cross on that grid. Now, you, in turn, must copy my cross which you see on your screen, onto the grid on your paper. And so it goes. Because the two visual spaces are different, they can't be shared.

The Shared Visual Space.

A shared visual space is one where the people at both ends see the same thing at all times, and both have equal access to altering what they both see. In order to initiate an interrupt, they must be able to input to the common viewing screen simultaneously. They must share both the seeing and the creating or altering, just as they do on the telephone, when they can, at the same time, both talk and listen.

Today's technology allows the creation of various forms of shared visual space. We can share a graphics space wherein we both see the same things, and can both input into that space. However, should our inputs collide, we must not react as computer purists and strive to protect the integrity of the individual data streams. NO! We must permit the streams to collide and produce what they will. An interruption is, after all, a change in the expected way the world is to unfold.

Also, as the shared visual space only makes sense when the two parties are also connected by telephone, any confusion, loss of synchronization etc., that might result, can be quickly resolved. Proper simultaneous voice-and-data shared space communication makes the computer scientist a bit uneasy. It relates to enhancing human interaction, not computer niceties. As the new technology got cheaper, and our ability to apply it improved, it became easier and easier to achieve simultaneous shared acoustic and shared visual spaces. Some spectacular teleconferencing experiences were generated this way. However, it was soon discovered that most people were not capable of operating in an unconstrained shared graphic space. Cartoonists had demonstrated what could be done, but for most people, the results more closely resembled the graphic outpourings of preschoolers.

The more proscribed visual spaces of word processing and electronic broadsheets were then examined. This work was started using Radio Shack Model I machines, back in the VTOS era. Versions of ELECTRIC PENCIL and VISICALC were prepared that exhibited many of the shared space characteristics. However, with the advent of Radio Shack's TRS-80 Model III, and Logical Systems' LDOS 5.1.2, with its superb communications and device filtering features, a truly workable and fully practical system could be easily produced.

The Reflex Connection.

In the course of this work, a new form of interconnection between two stand alone computers was defined. In deference to the mirror like symmetry that the particular connection method features, the term "REFLEX" was used to describe it. In operation, both machines scan their incoming data lines every time they scan the keyboard. If a locally generated keyboard character is available, they send that character to both the local CPU and to the outgoing data line. if a character is found in the RS232 incoming register, it is passed only to the ocal processor. In effect, both machines are fed from both keyboards, and respond equally to keystrokes from either. They are fully symmetrical, in a reflexive sense. Such a connection produces a shared visual space when the two machines are similar and are running similar software. If your TRS-80 is loaded with SCRIPSIT, suitably patched so as to use the LDOS keyboard driver, and the REFLEX filter is active, and if my machine is similarly configured, we can connect the two machines together via a data line and, while talking on the telephone to each other, jointly edit the same contract, or whatever. Whatever you do to your copy happens instantly to mine, and vice versa. And while this is happening, we are talking to each other about the changes we are making. We will arrive at a very powerful, democratically achieved consensus when we do this. With today's modems, this style of working requires two separate telephone lines, one for the telephone connection and one for the data connection.

With such an arrangement, two lawyers, separated by perhaps many thousands of miles, can confidently and very quickly, jointly produce or examine a contract, and so arrive at a mutually satisfactory one in a mere fraction of the time required if the mail, either electronic or conventional, were used. Press releases can be jointly perused, and a strong consensus built about the document, even though many miles may separate the communicants. It is far more meaningful to be able to see exactly what it is that is being discussed than to see the face of the person at the far end. Any significant information contained in the facial clues is also present in the aural clues, already available over the phone. The labels of Appendix I were composed using a Model I and a Model III running Scripsit reflexively.

However, our experience to date in demonstrating and eliciting usage of the reflexive way of working with another person suggests that although this might be the world's greatest mousetrap, few are trampling down our door. It is much too early to be certain about the causes of this resistance, but one hypothesis suggests that not only does our ability to learn new languages diminish significantly during our teens, but so does our ability to learn new modes of communications. If this is the case, then the big market for shared space or reflexive word processors must wait for a generation to come of age that has a different communications background, one that involves more sharing of information.

In the meantime, those who are at the leading edge, the explorers, can be encouraged to use the technique. Consequently, we have chosen to place the requisite software in the public domain, and publish in journals that have a readership most likely to apply the ideas of simultaneous voice and data shared space interaction between communicating micros. The LDOS operating system is, if not the only viable one, far and away the most suitable one, to support REFLEX.

Two appropriately programmed communicating personal computers can be so linked over a simple low speed data line as it is only the key strokes that get transmitted, and not the screen updates. The resident application package, present at both ends, be it SCRIPSIT, VISICALC or whatever, handles the demanding chore of updating the screens at both ends. Consequently, any low speed data facility, such as a 300 baud link, is very adequate. Perhaps, as familiarity with this shared space mode of working grows, and a market develops, smart modems specifically designed for this particular way of communicating may evolve. Such modems would eliminate the need for two telephone lines by sharing one line for both data and voice. As the data rate requirement of this mode of communication is extremely variable, ranging from an upper limit set by the need for speedy file transfer, down to a few keystrokes per second, the development of such apparatus, and the requisite interfacing drivers for the computers may not be easy.

Since it is only the keystrokes that are sent, non-similar machines using identical man/machine protocols can communicate. VISICALC, if properly modified, can run between an Apple and a TRS-80 in a fully reflexive, shared space way. The only difficulty is that the Apple lacks a full set of cursor positioning arrows, and needs some interfacing software to make up for this deficiency when connected to other machines with the full up/down, left-right arrow key complement on the keyboard. A "preloader" that conditions an Apple II to interpret the standard Apple VISICALC disk in such a way that VISICALC runs reflexively is now commercially available.

Vast differences in display sizes, such as that between the Apple and the TRS-80 are insignificant when the subject matter is viewed on a full screen editor that is driven by cursor position. The interest area is always where the cursor is, and not at the other side of the screen, where the display disparencies cause different material to be seen. In the case of the 40x20 display of the Apple and the 64x16 display of the TRS-80, the discrepancy was essentially unnoticed in operation.

The REFLEX filter, when active, allows any application package that runs on the TRS-80, and uses the system keyboard driver, to be run reflexively with another similar machine. Model I machines can work with Model III machines, for LDOS makes them sufficiently similar. As LBASIC, the LDOS version of BASIC, uses the system keyboard driver, anything written in BASIC, that does in fact use the system keyboard driver, can be run reflexively.

Prosoft's current version of NEWSCRIPT, being largely written in BASIC, uses the system keyboard driver for the majority of its commands, and so can be made to operate quite well with REFLEX. The BASIC code must be modified somewhat, however. The Prosoft driver, NS/CMD, must be placed at the top of memory as it is not fully relocatable code. The various BASIC programs look at the Keyboard DCB to determine the scratch memory's location.

However, once the LDOS keyboard driver and the REFLEX and MINIDOS filters have been put in place, the DCB no longer points to where it was expected to point. By setting the particular variables involved to -1765, the problem is avoided. Once the LDOS driver is installed, the overflow error message indicates this problem, and identifies the line that needs fixing. Peeks to &H4016 and 7 are the culprits. The final result is a very powerful word processor capable of fully communicating with another of its ilk in full REFLEX, of working alone, or of working into a large IBM mainframe system. This combination is the essence of a communicating micro!

Installing the LDOS keyboard driver somewhat alters the controls for NEWSCRIPT. The <CLEAR> key, which was used for control, now becomes <SHIFT><CLEAR>, with the <SHIFT> pushed slightly ahead of the <CLEAR> key. Only the toggle mode of control character entry remains, the continuous mode having been lost. It doesn't use the system keyboard call, apparently. If the <CLEAR><SHIFT> keys are depressed, with the <CLEAR> key being a bit ahead of the <SHIFT> key, then REFLEX and MINIDOS are addressed. Finally, where the <BREAK> key was previously used to break a line, <SHIFT><DOWN ARROW><A> now does this. The <BREAK> key returns control to LBASIC. Although it might seem that the controls are a bit more awkward, only the delete a bunch of characters is significantly slowed. Other techniques, such as breaking the line, and deleting to the end of the line make up for it. At the time of writing, a bug was noted in the printer driver portion of NS that becomes active when any keyboard driver other than NS/CMD is used. This bug persists even when the DCB path is "bent" so it loops into NS and then into either the regular or the LDOS keyboard drivers.

The REFLEX filter allows the TRS-80 to run in three modes, the conventional LOCAL mode, the REFLEX mode, and a simple terminal mode. These modes are chosen by selecting "I","U" or "Y" respectively, while the <CLEAR><SHIFT> combination is depressed. The more logical "R" and "T" keys are used by LDOS's MINIDOS, and so were not available.

REFLEX needs an eight bit data word as the LDOS keyboard driver can generate codes that need this longer word length. Consequently, to use REFLEX, it is necessary to first install the LDOS keyboard driver with the SET command, and then to install the proper RS232 driver with the 8 bit word option. The REFLEX filter can then be installed. The LDOS filter MINIDOS can then be installed, and the machine's configuration captured. The assembler listing for REFLEX/FLT for the TRS-80, Model III is in Appendix 1 with the changes noted for Model I.

The Simultaneous Voice and Data, Shared Space way of working together increases the need for easy ways to toss files back and forth between the communicants. Most application packages, of the word processing and broadsheet class, do not cater to sending or receiving the current work space. Two notable exceptions are VISICALC for the TRS-80 Model I, and Prosoft's NEWSCRIPT. Because the LDOS operating system, in SYS2, recognizes devspecs, like *SI, *KI etc., and substitutes the appropriate device for the usual disk drives and a filespec, NEWSCRIPT achieves this desirable feature, with effort on Prosoft's part. SCRIPSIT, unfortunately, clobbers the * in the Special Command area, and so can't send the contents of its work space in this easy fashion. This shouldn't be too hard to fix, and perhaps someone will locate this constraint and remove it.

VISICALC is rather special with regard to sending and receiving its work space. The feature is mentioned on the summary card, not in the instruction book. However, substituting :R for a filename, as is indicated on the instruction card, fails to produce the desired results. The code is simply incorrect in the Model I version. With a few fixes, however, it can be made to work. This feature even further enhances the value of VISICALC. Details of the fix are in Appendix II.

If the principal utility of the serious microcomputer is in the broadsheet and word processing areas, then Prosoft's NEWSCRIPT and the LDOS patched version of Model I VISICALC provide a means of doing these important things in a fully reflexive way. The Model III machine, because of its interrupt driven RS232 interface, does a somewhat better job than the older Model I. Also, the Model I seems constrained to booting in single density if NS/CMD is to be at the top of memory.

Interactive games, played in simultaneous voice and data in ways that are perhaps more complex than simple REFLEX, is a whole new area of possibility. A tic-tac-toe game, in BASIC, is shown in Appendix III.

This game uses the computers to handle the displays while the two players, connected via telephone and a data line, supply the logic. The computers act as referees and accept only legal moves. This game requires LDOS with the keyboard driver active, *SI active, and the REFLEX filter operating on *KI. This is a trivial game, but it does illustrate the notion of games in silmultaneous voice and data by using two communicating' micros in REFLEX.

One very popular current management theory suggests that the levels of mutual trust that exist in North America may be too low for effective human interaction. Enlarging the size of the common information space that communicants share is one way of encouraging the building of mutual trust. The trust level and the ease with which we adapt to more sharing of our information are likely quite symbiotic. Increases in one feed the other. As the technology gets more and more complex, perhaps we need to develop means of increasing the intellectual synergy between managers rather than increasing the competition. Learning to handle simultaneous voice and data shared space communications with our Communicating Micros is one giant step towards such an objective.

EDITOR'S NOTE: The EQUate listings below contain the differences between the Model I and III. Use whichever version is appropriate.

APPENDIX I

```
00100 ;RELOCATABLE REFLEX.
00110 ;By Jim and Gordon Thompson.June, 1982.
00120 ;
00130 ;                MOD 3   MOD 1
00140 ;
00150 @ABORT  EQU     4030H
00160 DODCB$  EQU     401DH
00170 @DSPLY  EQU     4467H
00180 @EXIT   EQU     402DH
00190 @GET    EQU     0013H
00200 HIGH$   EQU     4411H
00210 KIDCB$  EQU     4015H
00220 KIJCL$  EQU     42BEH   ;43BEH
00230 @PUT    EQU     001BH
00240 RFLAG   EQU     4413H   ;401AH
00250 SFLAG$  EQU     442BH   ;430FH
00260 SIDCB$  EQU     42C8H   ;43C8H
00270 ;
00275         ORG     5200H
00280 ENTRY:  PUSH    DE                ;SAVE DCB
00290         LD      A,(SIDCB$)        ;*SI DCB ADRESS
00300         BIT     03H,A             ;CHECK OFF/ON
00310         JR      Z,GOOD            ;*SI HAS DRIVER
00320         LD      HL,NOSI           ;"DUMMY!"
00330         CALL    @DSPLY            ;NO *SI MSG TO *DO
00340         JP      @ABORT            ;BACK TO SYSTEM
00350 GOOD:   LD      HL,BANNER         ;REFLEX TITLE
00360         CALL    @DSPLY            ;DISPLAY IT.
00370         LD      BC,LAST-START     ;PROPER LENGTH
00380         LD      HL,(HIGH$)        ;PUT INTO MOD III
00390         INC     BC                ;HIGH$.
00400         XOR     A
00410         SBC     HL,BC
00420         LD      (HIGH$),HL        ;NEW HIGH$ FOR MOD3
00430         POP     IX                ;RECALL DCB ADDRESS
00440         INC     HL
00450         LD      A,(SFLAG$)        ;SFLAG$
00460         BIT     5,A               ;DO in effect?
```

```
00470          JR      Z,NODO
00480          LD      IX,KIJCL$         ;KIJCL$
00490 NODO:    LD      A,(IX+1)
00500          LD      (START+1),A       ;ADDRESS LOADING
00510          LD      (RESTRT+1),A
00520          LD      A,(IX+2)
00530          LD      (START+2),A
00540          LD      (RESTRT+2),A
00550          LD      (IX+1),L
00560          LD      (IX+2),H
00570          EX      DE,HL
00580          LD      HL,START          ;COPY UP UNDER NEW
00590          LDIR                      ;TOP OF MEMORY
00600          XOR     A
00610          LD      (RFLAG),A         ;SET REFLEX BYTE TO
00620          JP      @EXIT             ;ZERO.GO TO SYS.
00630 ;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
00640 ;XX              ACTUAL FILTER
00650 ;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
00660 ;
00670 START:   CALL    0H                ;CALL TO DRIVER
00680 INIT:    PUSH    AF
00690          OR      A                 ;KEY PUSHED?
00700          JR      Z,ABORT           ;NO, GO AWAY!
00710          CP      080H              ;CHARACTER GTR 80H?
00720          JR      C,ABORT           ;PASS CHAR CALLER
00730          AND     0FFH
00740 RTEST:   CP      0F5H              ;TEST FOR "U"
00750          JR      NZ,LTEST          ;  SHTF & CLR.
00760          POP     AF
00770          LD      A,01H
00780          LD      (RFLAG),A         ;IN REFLEX,SET FLAG
00790          LD      A,52H
00800          LD      (3C38H),A         ;PUT "R" TO SCREEN
00810          XOR     A
00820          JR      INIT
00830 LTEST:   CP      0E9H              ;TEST FOR "I"
00840          JR      NZ,TTEST          ;  SHFT & CLR.
00850          POP     AF                ;IN LOCAL, SO
00860          LD      A,4CH             ;WRITE "L" TO
00870          LD      (3C38H),A         ;  SCREEN.
00880          XOR     A                 ;SET A REG TO ZERO.
00890          LD      (RFLAG),A         ;SET REFLEX FLAG
00900          JR      INIT              ; TO ZERO
00910 TTEST:   CP      0F9H              ;TEST FOR "T"
00920          JR      NZ,ABORT          ;PASS CHAR TO CALLER
00930          POP     AF
00940          PUSH    DE
00950          PUSH    IX
00960          LD      A,54H             ;WRITE "T" TO
00970          LD      (3C38H),A         ;  SCREEN
00980 REINIT:  LD      IX,KIDCB$
00990          LD      DE,KIDCB$
01000 RESTRT:  CALL    0H
01010          OR      A
01020          JR      Z,SI
01030          CP      0E9H              ;IS IT CLR SHFT I?
01040          JR      Z,REPAIR
01050          CP      0F5H              ;IS IT CLR SHFT U?
01060          JR      Z,REPAIR
01070          CP      0F9H              ;IS IT CLR SHFT T?
01080          JR      NZ,CONT           ;YES.
```

```
01090          JR      RESTRT          ;  GO BACK.
01100 CONT:    PUSH    AF
01110          LD      DE,SIDCB$       ;*SI DCB ADDRESS
01120          CALL    @PUT            ;OUTPUT TO SI
01130          POP     AF
01140          PUSH    AF
01150          LD      DE,DODCB$       ;*DO DCB ADDRESS.
01160          CALL    @PUT            ;WRITE CHAR TO *DO
01170          POP     AF
01180 SI:      LD      DE,SIDCB$       ;*SI DCB ADDRESS
01190          CALL    @GET            ;INPUT FROM *SI
01200          OR      A
01210          JR      Z,REINIT
01220          LD      DE,DODCB$       ;*DO DCB ADDRESS
01230          CALL    @PUT            ;WRITE TO *00
01240          JR      REINIT          ;END OF "T" LOOP
01250 REPAIR:  POP     IX
01260          POP     DE
01270          JR      INIT
01280 ABORT:   LD      A,(RFLAG)       ;SEE WHAT MODE WE IN
01290          OR      A               ; 1 FOR REFLX, 0 LOC.
01300          JR      Z,LEAVE         ;IT'S LOCAL
01310          POP     AF              ;IT'S REFLEX
01320          PUSH    AF
01330          OR      A
01340          JR      NZ,TALK         ;KEY, GO TO *SI
01350          POP     AF              ;NO CHARACTER.
01360          LD      DE,SIDCB$       ;*SI DCB ADDRESS.
01370          JP      @GET            ;GET IT.
01380 TALK:    PUSH    DE
01390          LD      DE,SIDCB$       ;*SI DCB ADDRESS
01400          CALL    @PUT            ;OUTPUT *SI.
01410          POP     DE
01420 LEAVE:   POP     AF
01430 LAST:    RET                     ;RETURN TO CALLER WITH CHAR.
01440 NOSI:    DB      ' (((((( SI DRIVER NOT ACTIVE ))))))'
01450          DB      0DH
01460 BANNER:  DB      '*** REFLEX FILTER *** ',0AH,0AH
01470          OB      'June 1982  Version.',0AH
01480          DB      '<CLEAR SHIFT> I LOCAL',0AH
01490          DB      '<CLEAR SHIFT> U REFLEX',0AH
01500          DB      '<CLEAR SHIFT> Y TERMINAL'
01510          DB      0DH
01520          END     ENTRY
```

This is the BINHEX listing for the REFLEX filter. The checksum is C6.

```
D5 3A C8 42 CB 5F 28 09 21 F0 52 CD 67 44 C3 30 40 21 14 53 CD 67 44 01 98 00 2A 11 44
03 AF ED 42 22 11 44 DD E1 23 3A 2B 44 CB 6F 28 04 DD 21 BE 42 DD 7E 01 32 58 52 32 9B
52 DD 7E 02 32 59 52 32 9C 52 DD 75 01 DD 74 02 EB 21 57 52 ED B0 AF 32 13 44 C3 2D 40
CD 00 00 F5 B7 28 76 FE 80 38 72 E6 FF FE F5 20 0E F1 3E 01 32 13 44 3E 52 32 38 3C AF
18 E4 FE E9 20 0C F1 3E 4C 32 38 3C AF 32 13 44 18 D4 FE F9 20 4A F1 D5 DD E5 3E 54 32
38 3C DD 21 15 40 11 15 40 CD 00 00 B7 28 1E FE E9 28 2B FE F5 28 27 FE F9 20 02 18 EC
F5 11 C8 42 CD 1B 00 F1 F5 11 1D 40 CD 1B 00 F1 11 C8 42 CD 13 00 B7 28 CC 11 1D 40 CD
1B 00 18 C4 DD E1 D1 18 86 3A 13 44 B7 28 14 F1 F5 B7 20 07 F1 11 C8 42 C3 13 00 D5 11
C8 42 CD 1B 00 D1 F1 C9 20 28 28 28 28 28 28 20 53 49 20 44 52 49 56 45 52 20 4E 4F 54
20 41 43 54 49 56 45 20 29 29 29 29 29 29 0D 2A 2A 2A 20 52 45 46 4C 45 58 20 46 49 4C
54 45 52 20 2A 2A 2A 20 0A 0A 4A 75 6E 65 20 31 39 38 32 20 20 56 65 72 73 69 6F 6E 2E
0A 3C 43 4C 45 41 52 20 53 48 49 46 54 3E 20 49 20 4C 4F 43 41 4C 0A 3C 43 4C 45 41 52
20 53 48 49 46 54 3E 20 55 20 52 45 46 4C 45 58 0A 3C 43 4C 45 41 52 20 53 48 49 46 54
3E 20 59 20 54 45 52 4D 49 4E 41 4C 0D
```

APPENDIX II.

```
**   VISICALC PATCH TO FIX RS232C FUNCTIONS **
MODIFICATIONS TO LDOS PATCHED VISICALC.

.Hardware differences between MODEL I & III force
.the use of calls to 1B & 13 for RS232C in and out.
.RS232C Interface MUST BE PRESET TO DESIRED STATUS.
.
.Code at following address previously set the RS232C
.hardware when a ":R" was detected. This is now much
. reduced as a consequence of using the system calls.
X'9B3C'= 28 05 FE 50 CA 8B 9B B7 C9
.
.This code senses MODEL I OR III and loads DE with
.*SI DCB address.  It will be called during both SEND and
.RECEIVE operations.
X'9B45'= F5 3A 25 01 FE 49 28 05 11 C8 43 18 03 11 C8 42 F1 C9
.
.OUTPUT A BYTE TO *SI
X'9B7C'= CD C0 9B D5 CD 45 9B CD 1B 00 D1 B7 C9 00 00
.
.KEYBD SCAN, REPAIRED HANDSHAKE AND RECEIVE A BYTE
X'9751'= FA D5 CD 45 9B CD 13 00 D1 C2 57 9B CD 2B 00 28 F0 3E 60 37 C9 B7 C9 00 00
.
.CHECKS INCOMING BYTE FOR HANSHAKE
X'9B57'= FE 5B 28 02 B7 C9 3E 60 37 C9
.               **  NOTICE  **
.Although VISICALC expects a &H5B to terminate the RECEIVE
.state, the transmit mode never sends it.  Code at 9BC0 is
.never triggered into doing it, as the required &H60 is
.never there, so it never sends the &H5B.
.
.Incidently, in the original code, the send handshake was
.&H5B and the expected receive one &H5E!  We chose the &H5B
.for both.  They are located at 9BC5, and 9B58 as a result
.of the patch.

.Because the transmit mode doesn't automatically trigger
.the termination of the receive mode, the RECEIVE code has
.been altered to also accept any keyboard entry as a
.request to terminate the reception of incoming data, and
.return to normal VC.
.
.Should someone else be more successful at making the
.transmit send the proper termination signal, the receive
.routine, as it now stands, will terminate properly, as
.it contains the requisite code to terminate the receive
.function upon the receipt of &H5B.
.
```

APPENDIX III

EDITOR'S NOTE: Due to the length (3 pages) of the Tic Tac Toe listing, it was not printed in this issue. A copy of the listing may be obtained by sending a large, self addressed stamped envelope (postage will be approximately 37 cents.)

BASIC and File Structure - A Beginner's View

by Wes Goodnough

This article is written by Wes Goodnough, a relatively new "microcomputer" user, although he has experience on larger machines. It covers the subjects of file structure and business use of a microcomputer.

Since for a long time microcomputers were marketed principally as a hobby and personal product, software production had tended to focus on entertainment. More recently, however, the business world has begun to recognize the advantages of "a computer on my own desk that will give me the information I need to have." More and more micros are appearing on management desks, and sometimes in spite of the wishes of professional people in the "Department of Data Processing". In response to this growth, software offerings for business usage are also burgeoning. If the TRS-80 Model III is to have a place in business usage, it is operating systems software of the quality and scope of LDOS that will make it a contender in this market.

While entertainment programs have tended to focus on getting the most out of a computer in terms of manipulating its displays and in terms of interacting creatively with the hobbyist-operator, business oriented software must have other concerns. Indeed, when you come right down to it, the reasons for the proliferation of computers in our society in the first place have been business directed. The traditional rational for computer usage in business organizations is that they contribute to reducing the "cost of doing business", that they help an organization grow strong by providing more timely and accurate reports, and that they assist in insuring the basic satisfaction of customers doing business with the firm. As such, it is the DATA to be processed itself that is of concern to a business, not the creativity of the display nor the "elegance" of the programming. Happily, LDOS stands out as providing many enhancements that complement this concern.

Without a doubt, the VERSATILITY provided by LDOS is a major asset for business usage of the Model III. The availability and flexibility of drivers and filters as well as the JCL features within LDOS hold great opportunity for business application as well. And the device independence features of LDOS allow the use of high capacity storage devices. Above all, however, business data processing applications must be able to organize, store, retrieve, and manipulate literally millions of bytes of information, and an operating system that lends itself to this task is a must. For those who contemplate using a Model III in their business or for those who are intent on producing productive business software for the Model III, LDOS with its file friendly features stands out as a major resource.

It is my intent in this article to outline the basic file structures used in business data applications and to note some possible ways they may be implemented using the features of LDOS and LBASIC.

SEQUENTIAL FILE ORGANIZATION

The sequential file structure is the native structure of files on magn0 ic tape where processing must start at the beginning of the tape and proceed throuqh one record after another to completion. It is fundamental therefore, that with sequential file structure, even on direct access media, that the only sure entry point is at the beginning of the file. In addition, it is not hard to visualize that because new data cannot be inserted between records already written on the media, it is necessary in updating sequential files to re-write the entire file. Preferably this will be done on a separate drive in order to maintain at least two versions of the file. The term "Father-Son Processing" refers to the practice of maintaining a rotation of separate disks (usually three) to accomodate the update and security back-up of sequential files. It is inherent in sequential updating that the processing be batch oriented, for it is most efficient to make as many changes to the file as possible on any pass through it.

While sequential file structure does not readily lend itself to interactive applications, it should not be assumed that there is no place for sequential files in Model III applications. Indeed certain cases are actually improved with sequential access methods. In the case where all or nearly all of the records on file are used during processing, it is both efficient and orderly to proceed through the file from beginning to end. Such is the case in payroll applications where every record is examined and processing ought to proceed in an orderly fashion so as to provide adequate controls over the results. The strict order of records in a sequential file can also serve to control the order of other processing. Such was the case in an Election Returns System with which I am familiar. In order to produce a periodic printed report of the progress of precinct returns on election night, a file of candidates and proposals was prepared. Each pass through this file presented in fixed order the pertinent information to enable access to other files and tables where the vote tabulation was stored, and to format and print the desired report.

LBASIC supports the Disk BASIC of TRSDOS and uses the same structure and nomenclature for sequential files. Output to files is through the PRINT# instruction while input is through the INPUT# and LINE INPUT# instructions. Using these instructions, record formatting must be handled entirely by the programmer by the construction of the string variable argument used with these instructions. Concatenation of separate strings is one method of building the string arguement. If the programmer is careful to provide for a constant field length as well as for left or right justification and zero or space filling in each field, he will be able to extract the information accurately from the record when it is read from the magnetic file at a later date. To do this he will use the string functions LEFT$, RIGHT$, and MID$ as in the following example.

```
100 A$ = "TRAN"
110 INPUT B$: IF LEN(B$) > 15 THEN 110   ' a real
115 IF LEN(B$) = 15 THEN 130             ' hassle to
120 B$ = B$ + STRING$(15 - LEN(B$)," ")  ' get record
130 INPUT C: IF C > 999 THEN 130         ' set up for
140 C$=STR$(C):C$=RIGHT$(C$,LEN(C$) - 1) ' output to
150 C$=RIGHT$("000" + C$,4 - LEN(C$))    ' media
160 R$ = A$ + B$ + C$
170 PRINT #1, R$
180 ................................
....................................
....................................
790 ................................
800 INPUT #1, R$                         ' also a hassle
810 AS = LEFT$(R$,4)                     ' to retrieve
820 B$ = MID$(R$,5,15)                   ' data on file
830 CS = RIGHT$(R$,4): C = VAL(C$)       ' via string
840 '...............................     ' functions
```

By far the outstanding feature of LDOS as it pertains to sequential file handling is the expanded file OPEN Modes provided by LBASIC. In business data processing the data filed on magnetic media represents an almost priceless asset to an organization. The loss of a siqnificant file could be disastrous for even a healthy firm. By providing for "ON" (open new file at beginning), "OO" (open existing file at beginning), "EN" (open new file for extension), and "EO" (open existing file for extension), the programmer and the systems analyst can help prevent data loss due to operator errors. In updating sequential files, the "Father" file should be opened as "OO" insuring that the file is present where it is expected before processing begins, and that the file pointer is located at the beginning of the file. Then the "Son" file should be opened as "EN" or as "ON", likewise insuring that no other version of the file is present. An error returned on these entries would alert the operator to the possibility that a wrong disk was mounted and that data was about to be over-written.

SEQUENTIAL PROCESSING UNDER RANDOM MODE

For practical purposes, distinctions should be made between the term "Sequential" as a file mode and "Sequential" as an access method. On the one hand, the term is used to signify the file mode as distinguished from "Random", (viz; OPEN "O", 1, "FILE"). On the other hand, "Sequential file access method" may reasonably refer merely to the procedure of working through a file from beginning to end. Once this distinction is understood, new doors are opened to the programmer for he will understand also that by opening his files as "Random" mode files, ("RO", "R", "RN"), he may preserve the advantages of sequential access processing and still take advantage of the "Random" instruction set, (LSET, RSET, FIELD, MKI$/CVI, etc.).

Sequential processing under the "Random" mode is simple enough to be quite obvious. A FOR-NEXT loop is designed with the FOR-NEXT variable as the LRN of the GET instruction. Foremost among the advantages of the "Random" mode is the simplicity of constructing and interpreting the data record. The FIELD statement allows for identifying fields within the file buffer itself, both in terms of field tags and field lengths. Using the blocked file mode of LBASIC, logical records can be defined with sizes from 1 to 256 characters without the programmer having to distinguish between the various logical records within one physical record. Only one record is defined by the FIELD statement, and LBASIC extends that through the file buffer as needed. Once a record is accessed, any part of the record may be referenced by simple use of the string variable tag assigned by the FIELD statement. Maximum flexibility is provided by the ability to "re-define" the record through succesive FIELD statements. Likewise in formatting the record for output or in making changes to a part of the record, the FIELD string variables simplify access to the part of the record under immediate consideration. The LSET and RSET instructions not only assign values to the FIELD variables, but they also right or left justify the string within the "field", (including space filling). As for "zero filling" of numeric fields, the process is not required at all since numeric values are converted to strings representing the Binary configuration of the numeric value by the MKI$, MKS$, and MKD$ functions. Re-conversion is done with CVI, CVS, and CVD functions. The following is an example of the use of these "Random" features in sequential processing. Compared to the string manipulation in the previous example this is straightforward.

```
50  OPEN "RO", 1, "FILE",21     ' Open as Random
60  FIELD 1, 4 AS A$, 15 AS B$, 2 AS C$
70  ...................
90  ...................
100 FOR R  1 TO LOF(1)          ' Get one after another
110 GET 1, R                    ' from begining to end
......................
260 NEXT R
270 ...................
......................
540 ...................
550 LSET AS = "TRAN"            ' No fuss assignment of
560 LSET B$ = B1$               ' of fields - Correct
570 LSET C$ = MKI$(C)           ' lengths and justified
580 PUT 1, LOC(1)               ' Only one disk needed
590 ...................
```

While the sequential file mode may not be frequently used in business applications for the Model III, it is certain that "sequential access methods" are important tools to be understood and used, even under "Random" mode.

DIRECT ACCESS FILE ORGANIZATION

Once the programmer understands the use of the "Random" mode instruction set he will see the facility it offers in manipulating data in files. However, the subject of "Random" files is much broader than consideration of the FIELD and LSET instructions. Disk storage devices are considered as "Random" or "Direct" access devices, meaning that there is the ability to go directly to the record you want and read it without having to search sequentially from the beginning of the file. We know that LBASIC will allow us to specify the LRN of any record on file and, using the GET instruction, to pick out that specific record from the file. But how can WE know what LRN to specify unless we have the file organized in a way that will allow us to determine what it ought to be? This is the matter of file organization for direct access.

It may already be apparent to the reader that there is great confusion of terms in this subject. It seems that different manufacturers delight in using words in ways intended to distinguish their products rather than to communicate with a standard nomenclature. For purposes of this discussion "Random" will refer only to the LBASIC file mode as specified in OPEN "RO", or "R", or "RN". "Direct access" will refer to the general ability to go directly into a file and retrieve a specified record. Where any other meanings are intended for these terms, it will be clearly expressed.

THE RELATIVE FILE

There are basically only TWO ways to organize a file for direct access, though certainly there are many variations within each. The first to be considered is the RELATIVE file, (also sometimes referred to as a "direct" file.) The Relative file structure is made up of records which are stored in relative locations within the file. For instance, a file of the states of the Union would contain 50 records and each state could be assigned a key number between 1 and 50 to represent both its order of statehood and its relative position on the file. Thus the 18th record on the file would also be the 18th state admitted to the Union.

This plan is simple only so long as the key value does not exceed the maximum number of records in the file. Where the range of key values does exceed the number of records allocated, there must be some way to convert the key into a relative position number. One method for randomizing record placement is to divide the key by a prime number and use the remainder as the LRN. In the above example 47 would be the largest prime less than the file size of 50. For a key of 321, the calculation would be:

321 / 47 = 6 plus a remainder of 39

Hence the record with key 321 would be placed in relative position 39.

There are many such algorithms, each with its own properties and characteristics when used with a specific file. No matter what algorithm is chosen, it should ideally result in deriving LRNs evenly distributed over the range of the file and with a minimum of duplications. (In the example, key numbers 274 and 368 would also derive the LRN of 39.)

In case of a duplicate LRN being derived by the randomizing algorithm, there must be a way to determine an alternate location for the record. Very often it is to simply begin searching subsequent locations on the file until an open spot is found. Since the derivation of duplicate LRNs is more likely to happen as the file approaches being full, it is helpful to allocate extra positions to minimize the amount of sequential searching that needs to be done.

The main advantage of the relative file structure and the randomizing algorithms is that it provides the very fastest access times. Usually only a simple calculation and one I/O to the storage media are needed to retrieve a record. Therefore this is the method to choose when speed of access is very important. However, trade-offs are necessary to maintain this speed. If there are many duplicate

LRN derivations, then access time will be slowed and the advantages lost. Therefore it is important to be sure that the algorithm used is adequate for the application. Leaving extra room on the file may be a necessary overhead, for even though the extra space will never actually be used, access times will be minimized.

The coding to implement the search in relative files is very simple. Whether it is to find a place to insert a new record or to find a record on file, the process is the same, a simple calculation and a GET instruction.

```
100 R = KEY - 47 * INT(KEY/47)    ' The calculation
110 GET 1, R                      ' The file access
120 IF U$ = "1" THEN GET 1:       ' Check if duplicate
       GOTO 120                   ' If so get another
130 LSET ALL$ = REC$              ' New rec into buffer
140 PUT 1, LOC(1)                 ' Write to file
150 PRINT "RECORD IS FILED"
.......................
.......................
200 R = KEY - 47 * INT(KEY/47)    ' The calculation
210 GET 1, R                      ' The access
230 PRINT "RECORD IS FOUND"
```

INDEXED SEQUENTIAL FILES

The second method of file organization for direct access is the Indexed file. The Indexed Sequential Access Method or ISAM file organization is the most common example of the structure. With ISAM, records may be ordered in the file in an ascending or descending sequence by whatever key is specified. In addition there is an index to the file containing the LRN of every record along with its key. Much the same as a book index indicates the page where a subject is mentioned in the book, this index indicates where in the file a specific record will be found. 8y first looking in the index for the key being sought, the LRN is found and with only one GET, the record is in hand. The idea is that searching through the index is much quicker than searching through the file on the disk. Indeed using a "Binary Search", a specific item can be found in an index of 1000 items in an average of only 6 looks, while for an index of 10,000 items it will take an average of 7 looks. If the index is in memory (as in a matrix), this will provide fast access, although not as fast as relative file access.

In updating the ISAM file, it is only necessary to add new records at the end of the data file, and insert the appropriate entry into the index. For deletions it is enough to remove the appropriate item from the index, and for changes no alteration to the index file is necessary.

While it is easy to understand ISAM file updating, putting it together is not quite so simple, for there are many considerations to complicate it. First is data file management. If the data file will be maintained in strict sequential order, then it is necessary to determine how this is to be done. How will new records be added between existing ones? Or should new records be added to the end of the file, relying upon the index to give the ordering? Should deletions be removed altogether from the file or only flagged as being deleted? Each of these questions must be considered from standpoint of the programming required as well as of the needs and requirements of the specific application.

Secondly, it must be determined just how to handle the index itself. No matter how the data file is managed, it is paramount that the index be kept in good order. After making alterations to the index it may be necessary to resort it. For this the CMD "O" feature of LBASIC is an asset in that an index contained in a string matrix can be sorted reliably, and quickly. Even a part of the matrix can be sorted if necessary. Other questions involve the management of the index. Should it be a file itself, or a part of the Data file, (maybe the first few sectors)?

Should it be read into memory for use, or should it be manipulated on disk? Or should the index be "hard wired" into the program? The alternatives here are all dependent at least in part by the size of the index file and the requirements of the application. A small index could be "hard wired" if the data file contents do not change. Regardless of the variations resulting from these considerations, there are certain basics to coding the file access. The Binary search is the key to it and might be like this example.

```
100 A=1                               ' Lowest LRN
110 Z=50                              ' Highest LRN
120 H=50                              ' Current high LRN
130 L=1                               ' Current low LRN
140 M=0                               ' Middle point
150 '  KY$ is the key being sought.
160 '  K$(x) is the matrix containing the index.
......................
200 H=Z : L=A
210 IF LEFT$(K$(H),2) = KYS THEN 500  ' Found if true
220 IF LEFT$(K$(L),2) = KYS THEN 600  ' Found if true
230 M = INT((H-L)/2):                 ' Mid point
       IF N = L  OR  M = H THEN 600   ' Doesn't exist
240 IF LEFT$(K$(M),2) = KY$ THEN 500  ' Found if true
250 IF LEFT$(K$(M),2) > LEFT$(K$(L),2) ' Make M into
       THEN H = M  ELSE L = M         ' the next H or L
260 GOTO 230                          ' Try again
270 ..................
......................
500 PRINT "KEY IS FOUND": STOP
600 PRINT "RECORD DOES NOT EXIST": STOP
```

FROM FILES TO DATA BASES

Once the use of indices is mastered, it is only a short step from matters of file organization to matters more resembling data base structure. By using the same principals of indexing by keys and LRNs it is possible to create multiple indices for one or more files, (these indices are referred to as "inverted lists"). Each index is based on a key from a separate field or fields of the record. As the various indices are sorted in appropriate sequences, several paths through the file are available. By one it might be possible to read through in alphabetic order on salesmen's names, on another by monthly sales totals. With a little imagination and the application to make it meaningful, it would not be difficult to combine characteristics of more than one file in a single index and relate data of one file to data in another.

Another technique that does not qualify as a file organization but that is useable for relating one record and file to another is that of the linked list, (the use of pointers). With just two characters (using the MKI$ function on a given LRL) it is possible to include in one record the information for access to another record on the same file or even a different file. Chained files incorporate two such fields, one to point to the prior logical record and one to point to the next logical record. With such a file, all new records are added to the end of the file, but the chaining pointers are set to locate the record in its proper file sequence according to keys. Likewise, deletions are executed by changing the appropriate pointers to bypass the deleted record.

Both linked lists and inverted lists offer advantages that are important in maintaining and accessing data files. In business applications, where the storage and retrieval of large amounts of data is the heart of the processing concern, the success of application programming will be largely determined by the effective and knowledgeable use of appropriate data structures. Whether using Relative files or Inverted lists or Sequential access methods, the Model III with LDOS as an operating system has a good potential in business applications.

LISP IMPLEMENTATIONS FOR THE Z80

by Lee C. Rice & Daniel J. Lofy

Lee Rice is an Associate Professor of Philosophy at Marquette University in Milwaukee, Wis., and Dan Lofy is a former student of his with a degree in computer science.

I. LISP AS A PROGRAMMING LANGUAGE

LISP is a LISt Processing language which is based upon John McCarthy's work on nonnumeric computation, first published in 1960. The first LISP system was implemented at M.I.T. and described in the LISP 1.5 PROGRAMMER'S MANUAL. Since that time LISP has become the language of choice for virtually all work in artificial intelligence, and has been implemented on a variety of mainframes. The host computer for LISP at M.I.T.
was the DEC PDP-10, but research work there soon led to the production of a new minicomputer, "The Lisp Machine", whose hardware took full advantage of LISP's computational advantages. The LISP Machine provides a complete LISP environment: operating system, interpreter, editor, compiler, and utilities, as well as a machine-LISP implementation of other programming languages such as FORTRAN --> all written in, and managed by, LISP.

To a considerable extent, LISP remains unique among programming languages, possessing a flavor and feel all of its own. In 1973 during an invited talk at the University of Texas at Austin, Jean Sammett said, "Programming languages can be divided into two categories. In one category there is LISP; in the second category, all the other programming languages!" Today experienced programmers would probably want to add APL to this first rather exclusive category. Many programmers argue that both APL and LISP provide ideal operating environments for the microcomputer. In the past two years no fewer than two versions of each have become available for the TRS80.

This transition from mainframe to mini/micro mentality is just beginning. In the old days, machines were expensive, CPU time was equally costly, and disk space was a precious commodity. Programming languages like FORTRAN and COBOL were designed to minimize on-line programming. Programmers would spend hours coding flowcharts and pseudo-code, then compile their source code, pick up a long list of error messages, and return to their desks for debugging. But minis and micros have turned these priorities upside down: computer hardware, CPU time, and disk space are now among the cheapest of commodities in data processing, and programmer time among the most expensive. The use of interpretive and highly user-interactive languages is a first step in the right direction. More and more enhancements are being made to BASIC, and other interpretive languages such as LISP and APL are also coming into their own.

There are still two frequently heard myths about LISP in the computer industry: that it is hard to learn, and slow with mathematical calculations. Both have a grain of truth. For those who are used to programming in highly rigid programming languages such as FORTRAN or PASCAL, LISP takes a lot of getting used to. While it is also true that LISP was originally devoted to nonnumeric symbol manipulation, and handled math only haltingly, it is also true that modern implementations of LISP have a number-crunching ability second only to APL (APL, after all, was created to do with numbers just what LISP was created to do with symbols in general).

In terms of the microcomputer environment itself, LISP can't be beat. It is oriented toward programming at a terminal with rapid response. An extensive literature of utility programs is currently available. LISP uniformity also provides a very powerful programming tool for the micro user, once you become acclimated to it. LISP functions and LISP data have the same form. One LISP function can analyze another, or even use another. Indeed, thanks to its use of recursion rather than iteration, and its dynamic allocation of variables, LISP functions or programs can be used to alter themselves. Most LISP implementations, for instance, come equipped with a LISP editor (usually called EDIT, and written in LISP). If the user wishes to enhance the editor with new operations, this can be done simply by typing "EDIT EDIT" - i.e., one can use the editor to edit the editor!

All of these powerful features are bought at a price: the user has to learn some new habits, and unlearn some old ones. For LISP users, the gains are well worth the price. In the next section, we'll provide a very brief overview of LISP structures; and, in the following two, we'll describe the two implementations of LISP available for the TRS80 (both compatible with LDOS).

II. ELEMENTS OF LISP PROGRAMMING

English structures are made up of words, and so are LISP structures, except that there are two kinds of words in LISP: atoms and lists. Atoms and lists are called, generically, S-expressions (SEXes). Atoms can be English or nonsense words, but should not BEGIN with a number; since most LISP interpreters take any atom beginning with a number as a number-atom. A list is formed by enclosing any number of atoms, separated by spaces, with parentheses. The null list - "()" - containing no atoms is also identified with the atom NIL. Unless told otherwise, the LISP interpreter accepts the first atom in a list as a FUNCTION, and proceeds to evaluate it in the environment specified by the rest of the list.

Some examples of obvious LISP numeric functions follow. LISP's responses are indented two spaces to distinguish them from user input:

```
(DIFFERENCE 3.874 2.161)
  1.713
(TIMES 9 3)
  27
(PLUS (QUOTIENT 6 3) (TIMES 2 (MAX 2 4 3)))
  10
(QUOTIENT 3 2)
  1.333333
```

As the last example makes clear, LISP interpreters capable of handling floating point arithmetic do all of the conversions for the user. All functions are prefixed to their arguments, and the interpreter evaluates innermost parentheses first; so there are no rules of precedence to remember.

The most important LISP nonnumeric functions are CAR (which delivers the first element of a list), CDR (which delivers the list without its first element), and QUOTE (which tells LISP NOT to evaluate a list). So:

```
(CAR (QUOTE(APPLE PIE)))
  APPLE
(CDR (QUOTE(APPLE PIE)))
  (PIE)
```

The function CONS takes two arguments to make a new list:

```
(CONS (QUOTE LISP)(QUOTE (TAKES SOME GETTING USED TO)))
  (LISP TAKES SOME GETTING USED TO)
```

The LAMBDA operator is a means of binding variables locally to a function, and new functions may be introduced using the LISP function DEFINE. So we could define the functions FIRST, SECOND, and THIRD to pick out elements of a list as follows:

```
(DEFINE ((
'(SECOND (LAMBDA (LIST) (FIRST (CDR LIST))))
'(THIRD (LAMBDA (LIST) (SECOND (CDR LIST))))
'(FIRST (LAMBDA (LIST) (CAR LIST))) )))
  SECOND
  THIRD
  FIRST
(THIRD '(KEEP UP WITH YESTERDAY))
  WITH
```

```
(SECOND '(POP GOES THE WEASEL))
  GOES
(FIRST '(LISP IS FUN))
  LISP
```

As this example also makes clear the single quote is a means of avoiding too many parentheses in most versions of LISP.

To bring out the power of LISP's recursive techniques, we now define the function FACTORIAL, which delivers the factorial of a given number. To do this we also need the LISP function COND, which is just the ordinary "IF" function of BASIC or FORTRAN. COND simply takes a list of SEXes, evaluates them from the beginning until it finds a True SEX, and then returns the current value of that SEX. We add two additional LISP functions: ZEROP returns T (for "true") if a number is zero, otherwise NIL (for "false"); and SUBi merely subtracts one from a number. Now here is our definition:

```
(DEFINE 'FACTORIAL (LAMBDA (NUMBER)
  (COND ((ZEROP NUMBER) 1)
  (T (TIMES NUMBER (FACTORIAL (SUB1 NUMBER))))))
```

To create this definition, all we needed to know was that, for any number N, the value of FACTORIAL (N+1) is just (N+1) times the value of FACTORIAL (N). So we define the value of FACTORIAL (0), and then the value of FACTORIAL (N) in terms of FACTORIAL (N-1); and let LISP take care of the bookkeeping:

```
(FACTORIAL (4))
  24
(FACTORIAL (PLUS (QUOTIENT 6 3) (DIFFERENCE 5 1)))
  6
```

If we want to save the value of an operation we can use the LISP function SETQ, which stores the evaluated second argument as the value of the first (nonevaluated) atom:

```
(SETQ CHARLIE (FACTORIAL (PLUS (QUOTIENT 6 3) (PLUS 1 0))))
  CHARLIE
CHARLIE
  6
```

Notice that, unlike LAMBDA (which binds locally), SETQ is a global binder:
CHARLIE is now available for later use.

For situations where iteration is simpler or more easily readable than recursion (or where the beginning user needs to rely on techniques learned from FORTRAN or BASIC), LISP will also provide full iteration facilities via its GO and PROG features. GO, like the BASIC "GOTO", transfers control to another program segment (which is a character atom rather than a number in LISP). PROG is the LISP program generator, which enables the creation of a sequertial series of functions, and passing of values from earlier to later segments. For programmers who have worked in ALGOL, it is worth noting that the syntax of LISP PROGrams is the same as that of ALGOL. PROG variables are initialized to NIL, evaluation proceeds from left to right and top to bottom, and transfer is accomplished via GO and the insertion of labels (which are not evaluated). The function RETURN of one argument forces exit from PROG with the value being the value of PROG, or (if none exists) NIL. Finally, since LISP itself is a LISP structure, it can be used to describe and evaluate itself. The standard LISP debugger TRACE is written in LISP, and used to debug user-created LISP PROGs or functions; and similarly for the LISP editor.

Lest it be concluded that LISP is all sweetness and light, a few weak points should be noted. First, LISP is traditionally quite weak in disk I/O; and it is typically only capable of accessing LISP-created data files or workspaces. Mainframe enhancements to micro versions of LISP have only begun to appear.

Secondly, the abundance of parentheses and the freeform syntax (no line numbers, and LISP ignores blanks) make some form of formatting desirable for the CRT (in order for the user to read and edit longer programs), and an absolute necessity for printed output. Here again, mainframe versions of LISP are very strong, and micro versions are typically quite weak. For the experienced LISP programmer, these weaknesses are not so serious: a prettyprinter for LISP can be written in LISP. But, for the LISP novice, these shortcomings are often painful.

We have only touched upon the bare bones of LISP programming here. A typical LISP implementation for micros will offer fifty or sixty built-in functions (including PRINT, READ, and a host of mathematical tools), debugger, editor, and a variety of separate utilities. LISP, like APL, is also much more economical on memory usage than BASIC or FORTRAN. On a 48K TRS80 I have never needed to chain programs because of failure to fit them into a workspace. Further, since LISP makes no distinction between data and programs, tape or disk access is usually quite rapid.

III. SUPERSOFT LISP

Supersoft LISP (available from Supersoft Associates, P.O. Box 1628, Champaign, Illinois 61820) is designed to be a standard LISP interpreter. Essentially it uses the original (and now somewhat dated) M.I.T. version LISP 1.05, with the following enhancements: full floating point math, disk or tape I/O, and generous numbers of PROG features for the novice programmer. Updated most recently in 1982, it is now known as LISP 3.71, and is available for TRS80 Model I (tape or disk versions) and TRS80 Model III (tape or disk versions), as well as CP/M. The Model III version is not the same as the Model I version; and the disk versions of each contain many disk enhancements over their tape counterparts. The Model III version is fully compatible with LDOS, and the Model I version is supposed to be also (the authors have not verified this).

In Supersoft LISP each SEX is evaluated as input, which results in a genuinely interactive environment, and a pleasant atmosphere for computing. The implementation uses an association list (ALIST) to keep track of variable bindings, and an object list (OBLIST) to keep track of functions currently in the workspace. The user has full access to the OBLIST (delete or add functions), but the interpreter also does bookkeeping; and an efficient garbage collection routine is also implemented. To prevent catastrophe, there is no user access to the ALIST, which is continuously updated by the interpreter.

Enhancements to LISP 1.05 are worth noting. CHR, PEEK, and POKE (same as the BASIC functions) are fully implemented. PRINT, PRIN1 (same as PRINT without carriage return), LPRINT, LPRIN1, READ (for user input at terminal: same as BASIC INPUT), and TERPRI (which outputs a single carriage return to CRT or printer) are also provided.

The disk versions (Model I and Model III) contain a BREAKpoint function for debugging, standard disk SAVE and LOAD functions, and a special disk-save function (SAVFNS) for saving multiple functions together in a single disk file. SAVFNS is of particular use where the user has created many short functions, since saving these together in a single disk file saves disk space and disk access time. The author has also generously provided a LISP library of sixty additional functions. These include such niceties as EXPT (exponentiation), RADIX (to change a number from base 10 to any other base), and GENSYM (which enables the user to generate new atoms). Three larger programs are also included on the disk. TRACE is a LISP debugging aid which enables the user to step through the execution of any user-created function or program. EDIT is a LISP editor (with full breakpoint facilities). The program DERIV is a mathematical one which takes the derivative of an argument with respect to a single variable, and (optionally) allows the user to generate a Taylor series for a given function. All of these are written in LISP, and full source code is provided in the manual. The manual also provides a complete explanation of the sixty-five error messages generated by the interpreter, and (for the advanced programmer) details of the machine representation of LISP.

The manual is NOT an introduction to LISP, so it assumes that the user does have some introductory materials or text on hand. Also a weakness is that the manual is not well organized. Apparently, following each update of LISP, the author simply added appendices. The result, for instance, is that there are no less than three versions of disk SAVE given, and only the last is correct. This fact, coupled with a complete lack of index, means that Supersoft would be well advised to prepare a completely new manual from scratch.

The interpreter lacks any provision for prettyprinting output, so lineprinter output is almost unreadable. The editor is also quite modest and limited. For the novice these are serious shortcomings, but the manual does indicate procedures for editing the editor, and a prettyprinter can be written. On the positive side for the novice programmer, the interpreter is generally tolerant if too few parentheses are added; and, while only the error numbers are generated by the interpreter, the error messages listed in the manual are clear and explicit. The OBLIST is also unaffected by an error condition; and, after such a condition occurs, the interpreter returns at once to input level. High memory cannot be protected when entering LISP, so the use of the LDOS drivers and filters is also inhibited.

Supersoft LISP is as powerful a micro implementation of LISP as we are likely to have for a long time. Full use of it does require homework for the novice LISP programmer, but one or two introductory books on LISP, coupled with some patience when confronted with many error messages, will open up a new world of programming power.

IV. UOLISP

UOLISP is a recent implementation of LISP, originating in 1980 and updated three times since (most recently in April of 1982), first created at the University of Oregon. Unlike Supersoft LISP, UOLISP is a subset only of Standard LISP, and does NOT include provision for floating point math. Its support utilities are, however, much more extensive than its Supersoft counterpart. These include the interpreter, a compiler for generating fast-load files or directly executable code, a program to load fast-load files, an optimizing phase for the compiler, a function TRACE feature, a structure editor and prettyprinter, another version of LISP called RLISP, and many support packages.

First, the interpreter. It supports integers (range -4096 to +4095), strings of up to 255 characters in length, fixed point numbers, and several additional LISP features not supported by Supersoft LISP (dotted pairs, code pointers). A supporting package (BIGNUM) implements arbitrarily large integers, and yet another package supports vectors of arbitrary size (including vectors whose components are vectors, i.e., matrices). In addition to the standard LISP functions and predicates, the UOLISP interpreter comes equipped with compiler support functions; and these are of particular use to experienced assembler programmers.

Unlike Supersoft LISP, UOLISP provides a complete compilation process, which is accomplished in two passes. The first of these translates LISP into a pseudo-assembly code called LAP ("LISP Assembly Program"), and the second translates LAP into absolute machine code, placing it into storage (for execution) or into a fast-load file for later reloading. Optionally, a third pass optimizes the LAP before assembling it. Fast-load files are both relocatable and implementation independent. They can also be executed directly from LDOS as standard /CMD files. The debugging process takes place at the interpretive level, which makes the LISP compilation procedure faster and more efficient than FORTRAN, COBOL, and most implementations of PASCAL.

The LISP compiler is superior to most other compilers because of the ability of LISP to manipulate LISP: the compiler is written in LISP, and compiled LISP functions run from 10 to 50 times faster than the same functions in the LISP interpreter (which already run at about 5 times the speed of BASIC). The LAP assembler is accessible to the user, so fast assembly language routines and output to nonstandard devices can be implemented. An adjunct package for the optimizer also displays LAP code in assembler format.

For new LISP programmers, the Structure Editor and Prettyprinter will be welcome additions. The former is a LISP program which permits entry of functions, execution, and then saving as a disk file. This disk file can be accessed by all LDOS utilities, and even read by BASIC programs. The Prettyprinter interfaces to the Structure Editor so that LISP structures are automatically displayed in indented and easily readable format.

There is an added bonus for the non-LISP programmer. The language RLISP (which is written in LISP, as if you didn't know already!) provides a palatable alternative to LISP which looks something like PASCAL. The language features WHILE loops, REPEAT loops, several different FOR loops, and infix operators for math. For example, in LISP the function NTH would be defined:

```
(DEFINE 'NTH (LAMBDA (LIST N)
  (COND ((EQUAL N 0)(CAR L))
  (T (NTH (COR L) (DIFFERENCE N 1))))) )
```

whereas in RLISP we have:

```
EXPR PROCEDURE NTH (LIST, N);
 IF N=0 THEN CAR L
 ELSE NTH (CDR L, N-1);
```

PASCAL users, or devotees of structured programming, will follow the RLISP program flow easily. For those of us (authors included) who are LISP fanatics, the suggestion that LISP could profit from the clumsy structuring of PASCAL amounts to little less than heresy. Since structured programming is, however, the current fad, RLISP meets one possible market demand.

If the programmer does not like PASCAL or structured programming, there is yet another alternative to LISP included as a utility. The little META translator is a very big LISP program which permits you to create your own programming language, specify its syntax and how it is to be interpreted, and (optionally) convert it into assembly code. The manual contains a short example in META of a program which simulates a pocket calculator, but there is in fact no limit to what META can do. Experienced programmers can even write their own FORTRAN compilers in LISP using META (if this sounds fantastic, we should note that it has already been done, and the results mark a considerable improvement over the outdated Microsoft implementation of FORTRAN IV!).

On the negative side, the absence of floating point math makes UOLISP seriously deficient. Unlike its Supersoft counterpart, the UOLISP manual is well organized and very concise. It too assumes considerable familiarity with LISP structures; and it contains many sections devoted to assembler programming. It should be noted that UOLISP files are NOT compatible with Supersoft LISP files. Because of the great command over file formatting provided by LDOS, the LDOS user may reformat many Supersoft LISP files for UOLISP input (provided they do not utilize floating point math!); but the path from Supersoft LISP to UOLISP is a one way street. The UOLISP package offers a solid implementation of LISP for any programmer. Its compilation options will appeal to the serious programmer, while its prettyprinter and utilities will be attractive to anyone. UOLISP runs under Model I or Model III LDOS, with a minimum of 32K (48K recommended), and requires at least one disk drive (two are recommended). It is available from Far West Systems and Software, Box 3301, Eugene, Oregon 97403.

V. CONCLUDING NOTES

LISP is an exciting language to learn, and a powerful language to use. It has the potential for offering a complete programming environment, and both versions take full advantage of both the Z80 and the TRS80.

Four books dealing with LISP are particularly noteworthy for the beginning "LISPer". LET'S TALK LISP, by Laurent Siklossy (NJ: Prentice-Hall, 1976), offers an easy-to-read introduction to the distinctive and most powerful features of LISP programming. LISP, by P. H. Winston and B. Horn (MA: Addison-Wesley, 1981), is a much longer book which provides both an introduction to LISP programming and information on the implementation of LISP on many computers. Ken Tracton's PROGRAMMER'S GUIDE TO LISP (PA: Tab Books, 1980) is just what it claims to be, and has many useful LISP programs (as well as many typographical errors). Finally, THE LITTLE LISPER, by Daniel Friedman (Chicago: Science Research Associates, 1974) provides the most light-hearted and fun-filled tutorial on LISP programming imaginable.

FORTRAN, COBOL and LDOS JCL

The following article by Glen Rathke demonstrates how to simplify the compiling of FORTRAN and COBOL programs through the use of LDOS's JCL.

Working with Radio Shack's or Microsoft's Fortran can be less painful if used in conjunction with the JOB CONTROL LANGUAGE found in LDOS. First use the EDIT/CMD module to produce a working program such as the TEMP/FOR example that is in the front of Radio Shack's manual.

Next use the LDOS command BUILD to create a JCL file by issuing a BUILD FORTRAN command at the LDOS Ready prompt. Then, type in each line and press <ENTER>.

```
%1F.   1 = COMPILE  and check for ERRORS
.   2 = COMPILE, generate code, EXECUTE
.
//KEYIN - PLEASE INDICATE YOUR CHOICE
//1
F80 =#FILE#
%1F.   3 = QUIT
.   4 = RECOMPILE generate code EXECUTE
.   5 = EDIT
//KEYIN - PLEASE INDICATE YOUR CHOICE
//3
//EXIT
//4
F80 #FILE#,#FILE#=#FILE#
L80 #FILE#-G
L80 #FILE#-N,#FILE#-E
//EXIT
//5
//EDIT
//EXIT
//2
F80 #FILE#,#FILE#=#FILE#
L80 #FILE#-G
L80 #FILE#-N,#FILE#-E
//EXIT
///
%1F. JOB aborted NOT A VALID CHOICE.
//EXIT
```

Then use the DO command; DO FORTRAN (FILE=TEMP)

A KEYIN response of 1 will check for errors while compiling the program TEMP/FOR which was created using the EDIT/CMD module. If any errors are found at this point you can still go back and make any necessary corrections when prompted at the second KEYIN prompt. If the program is error free, RECOMPILING and EXECUTION are still available as well as a QUIT command.

A keyin response of 2 will then take control of the three remaining modules (F80/CMD, L80/CMD, and FORLIB/CMD) and process the file TEMP/FOR that was created under the EDIT/CMD. The token in the JCL files #FILE# has to be set equal to TEMP, which is done in the DO command where (FILE=TEMP). Then anytime the token "FILE" is encountered it will be replaced with the program name TEMP. The important part here is that whatever the name of the Fortran program is given, that same name will have to appear in the DO command line. In order to get a complete listing of the TEMP program including the code generated by the compiler you can BUILD another JCL file which I call "PRINT".

```
LBASIC
CLS
LPRINT CHR$(140)
CLS
CMD"S"
LIST #FILE#/FOR (P)
LBASIC
CLS
LPRINT CHR$(140)
CLS
CMD"S"
LIST #FILE#/LST (P)
LBASIC
CLS
LPRINT CHR$(140)
CLS
CMD"S"
//EXIT
```

Once again use the DO command but this time execute it as DO PRINT (FILE=TEMP). Although the JCL file may be a bit cumbersome, the end result will have a complete listing as well as each listing starting on its own sheet due to the LPRINT CHR$(140) embedded under LBASIC.

One more use for the JCL file can be found when you are sure that your program is bug free and when you want to clean up your work diskette, a multiple KILL file could be activated.

```
KILL #FILE#/LST
KILL #FILE#/REL

DO KILL (FILE=TEMP)
```

This file will only leave you with the original TEMP/FOR and the TEMP/CMD, so be sure that you don't need the other two files (especially TEMP/LST which gives you the code listing generated by the compiler.) These JCL files and commands were created using the 5.1.3 revision.

As shown in a previous example with FORTRAN, the JCL function can be used with great ease when performing multiple or related functions. In the next example a JCL file is used to control the COBOL compiler marketed by Radio Shack, written by Ryan McFarland.

The purpose of this application is to let the operator "take control" of the various options that affect compiling of a Cobol program.

Notice the use of a non numeric character "E" in answer to a KEYIN (actually any non numeric character or a numeral >5 will obtain the same result.) This response will bypass all other options or KEYIN(s) and will directly start EXECUTing.

```
%1F.............................................................
.   1   Compile (list to Video)
.   2   Compile (list to Video & Printer)
.   3   Compile (list Errors only to Printer)
.   4   Compile (X reference to Printer)
.   5   Compile "Debug" source lines
.   E   EXECUTE
.............................................................
.
//KEYIN - YOUR CHOICE
//1
RSCOBOL #FILE# (T)
%1F.  1   QUIT
.  E   EXECUTE
//KEYIN - YOUR CHOICE
//1
//EXIT
//2
RSCOBOL #FILE# (T P)
%1F.  1   QUIT
.  E   EXECUTE
//KEYIN - YOUR CHOICE
//1
//EXIT
//3
RSCOBOL #FILE# (P E)
%1F.  1   QUIT
.  E   EXECUTE
//KEYIN - YOUR CHOICE
//1
//EXIT
//4
RSCOBOL #FILE# (P X)
%1F.  1   QUIT
.  E   EXECUTE
//KEYIN - YOUR CHOICE
//1
//EXIT
//5
RSCOBOL #FILE# (D)
%1F.  1   QUIT
.  E   EXECUTE
//KEYIN - YOUR CHOICE
//1
//EXIT
///
RUNCOBOL #FILE#
//EXIT
```

Use the DO command to execute the above JCL program. DO COBOL (FILE=CALCXMPL)

LDOS and a HAYES SMART MODEM

John Mullins talks about the Hayes Smart Modem and LDOS. From the number of calls we get, it appears that the Hayes modem is pretty popular.

LDOS and a HAYES SMART MODEM make a very good pair, and with a program to issue commands from DOS and a device filter to disable output to the modem until a carrier is received, and a small amount of linking, a host system is in operation. The modem will detect the presence of an incomming call, and if it finds a carrier signal on the other end - will set the carrier detect input of the RS-232.

Then if the *DO and *KI have been linked to the Comm Line (*CL), the caller (from miles away) can issue keyboard commands and see display output. This simple system begins with a command sending program, MODCMD. All it will do is allow you to give the modem a line of commands. It is assumed at this point that the RS232 driver is in place at the time MODCMD is used. For example, MODCMD AT T D 555-3344 <enter> will tell the modem to dial (using tones) 555-3344. The format of the line is as follows (the "'" characters are used to mark the separate fields within the command):

MODCMD '*device name <space>''digit delay <space>' command

'*device name' will default to *CL, and the 'digit delay' will default to 3. The delay will allow the user to see the modem's responses to the commands. The second program is a filter so the RS232 driver will be ignored until a carrier is detected. Without this filter, the display will be slowed down to the speed of the driver (probably 300 baud) even if no one is online. The modem will think that anything it receives when not online is a command, if it receives other characters it can lead to unpredictable results. The next listing is a small JCL file to implement these commands in the correct order. The modem commands can be any you wish, but they must be issued before the online filter, and the online filter must be in place before the links.

```
.reset modem to power-up configuration
MODCMD AT Z
.set up to dial with tones, ignore backspaces
.have a 2 second escape guard, answer on 5 rings
MODCMD 4 AT T S5=138 S12=100 S0=5
FILTER *CL ONLINE send output only if online
LINK *DO *CL copy output to *CL if online
LINK *KI *CL accept input from modem as kbd input
.ready to answer
```

```
00100         TITLE   '<Modem Command Output>'
00110 ;
00120 ;MODCMD will allow the user to send a line
00130 ;of commands to the Hayes Smart Modem. If
00140 ;desired, it will wait for a response for
00150 ;a variable length of time.
00160 ;
00170 ;format:
00180            MODCMD |*dev <sp>||delay <sp>| <command> <enter>
00190 ;*dev if not included will default to *CL
00200 ;delay will default to 2
00210 ;      if delay is 0 no wait will occur
00220 ;command is a command to the modem ex. AT Z
00230 ;
00240 ;            MODCMD 0 AT Z
00250 ;
00260         ORG     5200H
00270 *GET EQUATE1/EQU
00280 ;
00300 START   LD      A,(HL)          ;check input line
00310         CP      '*'
00320         JR      NZ,CL
00330         INC     HL
00340         LD      A,(HL)
00341         CP      'Z'
00342         JR      C,$+4
00345         AND     5FH
00350         LD      (DCB+1),A
00360         INC     HL
```

```
00370          LD       A,(HL)          ;different device
00371          CP       'Z'
00372          JR       C,$+4
00375          AND      5FH
00380          LD       (DCB+2),A
00390          INC      HL
00395          INC      HL
00400          LD       A,(HL)
00410 CL       CP       ':'             ;a digit?
00420          JR       NC,OPENS
00430          INC      HL
00435          INC      HL
00440          SUB      '0'-1           ;noramlize 1..10
00450          LD       (CNT+1),A
00451 OPENS    PUSH     HL
00460          LD       B,1
00470          LD       DE,DCB
00480          LD       HL,BUFF
00490          CALL     @OPEN           ;open dcb
00510          JP       NZ,@ERROR
00520          PUSH     DE
00530          POP      IX
00540          LD       D,(IX+2)
00550          LD       A,D
00560          LD       (POINT+2),A
00570          LD       E,(IX+1)
00580          LD       A,E
00590          LD       (POINT+1),A
00600          PUSH     DE
00610          LD       DE,TCB
00620          LD       A,8             ;task slot   8
00630          CALL     @ADTSK
00640          POP      DE
00645          POP      HL
00650 LOOP1    LD       A,(HL)
00651          CP       'Z'+1
00652          JR       C,OK
00653          AND      5FH
00660 OK       INC      HL
00670          PUSH     AF
00680          CALL     @PUT
00690          POP      AF
00700          CP       0DH
00710          JR       NZ,LOOP1
00715 CNT      LD       B,3
00716          JR       TST
00720 LOOP2    PUSH     BC
00730          LD       BC,0000
00740          CALL     @PAUSE
00750          POP      BC
00760 TST      DJNZ     LOOP2
00761          LD       A,8
00762          CALL     @RMTSK          ;REMOVE TASK
00770          JP       @EXIT
00780 TCB      DEFW     TASK
00790          DEFW     0000,0000,0000
00800 TASK     PUSH     DE
00810          PUSH     IX
00820          PUSH     AF
```

```
00830 POINT   LD      DE,0000
00840         CALL    @GET
00850         CALL    NZ,@DSP
00860         POP     AF
00870         POP     IX
00880         POP     DE
00890         RET
00900 DCB     DEFB    '*CL',0DH
00910         DEFS    32
00920 BUFF    EQU     $
00930         END     START
```

```
00100         TITLE '<ONLINE/FLT>'
00110 ;
00120 ;This will filter the @put such that the
00130 ;carrier of a distant modem must be present
00140 ;for output through @put. Note: @ctl will
00150 ;still send messages with or without cd set.
00160 ;
00170         ORG     5200H
00180 *GET EQUATE1/EQU
00190 ;
00200 START   PUSH    DE
00210         POP     IX              ;get dcb
00220         LD      H,(IX+2)        ;high byte of add
00230         LD      L,(IX+1)
00240         LD      (GO+1),HL       ;save in routine
00250         LD      HL,(4049H)
00260         LD      DE,LAST
00270         LD      BC,LAST-FIRST
00280         EX      DE,HL
00290         LDDR
00300         EX      DE,HL
00310         LD      (4049H),HL      ;save new mem top
00320         INC     HL
00330         LD      (IX+1),L
00340         LD      (IX+2),H        ;and driver add
00350         LD      HL,GMSG
00360         CALL    @DSPLY
00370         JP      @EXIT
00380 GMSG    DEFB    'Filter now inserted',0DH
00390 FIRST   EQU     $
00400         NOP
00410         JR      C,GO
00420         JR      NZ,GO
00430         PUSH    AF
00440         IN      A,(0E8H)        ;carrier det?
00450         BIT     5,A
00460         JR      Z,GO-1
00470         POP     AF
00480         RET
00490         POP     AF
00500 GO      JP      0000
00510 LAST    EQU     $
00520         END     START
```

CONFESSIONS OF A MACHINE LANGUAGE ADDICT

(or, Misery Loves Company,)
(or How to Convert Software to Run Under LDOS)

Ray Pelzer wrote the Visicalc patches published in the July '82 Quarterly. Here is his story on how those patches were developed, and instructions on fixing assembly language programs to run with LDOS and to take advantage of its different features. The actual patches are listed at the end of the article, including those for the new VC-160Y0-T83 version.

"Hi, what did ya do last night??"
"Oh, I started disassembling VisiCalc and..."
"W-H-A-A-A-A-A-T ???"

Yep, that's the kind of reaction I got when I started working on the patches for Model 3 Enhanced VisiCalc (LDOS Quarterly, Vol 1, No. 5). A lot of people would like to try the same thing, but are a little afraid to try their hands at it. I wasn't sure I'd have the nerve to finish once I'd started but, by applying the little "tricks" that I've picked up as time passed, it was much easier than even I expected. For that reason, I'd like to pass on some of those tricks to you, so that you can better understand machine language programming, and learn some of the classy techniques used by others, as well as help you customize machine language programs on your own.

Since there's no possible way to teach you Assembler and Machine Language in a magazine article, I'm going to have to assume you are already a "hacker" of sorts (or, as some people in Radio Shack prefer, "Billy Bytehead"). You'll have to be able to comprehend the machine language level of programming to a fair degree (preferably, you'll be able to recognize some instructions at a glance of the hex code for speed's sake), and be able to run Editor-Assembler and Disassembler programs. Of course, I've been extremely pleased with both the Misosys products, EDAS 3.5, and Disassembler 1.0, but on the latter, one small deficiency in the program is the inability to disassemble directly from a disk file - only from machine language in memory. Knowing what I know of Roy, though, he'll probably have a new version ready to do just that by the time you read this article. In the meantime, I keep an old copy of the Apparat disassembler (from my wet-behind-the-ears days) on hand for those special cases. For this article, I'll refer back to the work I did on VisiCalc, and explain HOW I did it.

Probably the most important thing you can get from a disassembly is the location cross-reference (usually abbreviated to cross-ref or xref). As the disassembler does its thing, it creates a table of memory locations it THINKS have been referenced by the program (more on THAT later) for display or printout. You'll absolutely NEED this table in a later step if you are modifying an existing program (to make sure you don't cause another part of the program to crash due to your changes).

After you get a printout of the cross-ref (the one I did for VC came out to 42 pages for the cross-ref alone!), you should also get a HEX/ASCII dump of the program, such as LDOS's LIST command using the (HEX,P) parameters. NOW, you'll see what I meant about what the disassembler THINKS are references to memory locations in the program. If you've ever looked at a disassembly that seems to have a lot of meaningless transfers from one register to another, with a few JR NZ,xxxx jumps thrown in, look at those memory locations on your hex dump. You'll most likely find that those areas are loaded with text, because the disassembler mistakenly translates the characters to the LD r,r' series of op codes that they match. As for the JR NZ stuff, those are the blank spaces between the words! Once you have all this sorted out, you can "whittle down" the cross-ref list by crossing out the meaningless references like those.

Next step - look for the obvious. In VC, I wanted to find out something about disk I/O, so I looked for all the standard references relating to disk I/O (4420H, 4424H, 4436H, 4428H, etc.). Also, take a close look at anything you don't recognize. This is how I found that VC was conking out at the RAMDIR call in TRSDOS 1.3, since in LDOS 5.1.2, that call to 4290H (Model I) was a trip into the la-la land of SBUFF$, the system buffer, and in the Model III version, the RAMDIR call had not been installed

until version 5.1.3. While you're at it, keep a close watch out for funny things like ROM calls. If you've ever tried to use a Model 3 ROM call that doesn't exist on a Model 1, you know what I'm talking about. In some cases, especially for short routines, it might be easier to just duplicate the subroutine directly into the program in order to make it transportable, but this will be a try-and-see situation.

Now, for the first few steps of the modification - looking for the proper value of HIGH$, and the keyboard input alteration. VC comes with its own keyboard driver, so once that was disabled, I had a nice chunk of free space for inserting the HIGH$ check. Here, the old LSI patches for Model 1 VC gave me a hand. The first thing I had to look for was a reference to 3801H, the keyboard's memory-mapped area. That found, I backed up to what looked like it could be the beginning of the subroutine - the first byte following a C9 (RETurn). Looking up THAT address in my cross-ref, I was pleased to find it's reference quickly, and in only one place. Now, I could change THAT call to a call to 002BH, the ROM keyboard call.

Next came the actual HIGH$ check. In the original program, the Mod 3 HIGH$ storage point of 4411H was referenced twice, both times in the first sector of the file. Those two references were swapped out for a call to the area where the old keyboard driver had been. Now, as a short-cut, I looked back to LSI's fixes for the basis. Later on, I had to add some extra goodies in this area during the check, because the disk I/O vectors I used were in different places on Model 1 and 3, and this was as good a place as any to do the switching. However, since that stuff was yet to be done, I did a shorter temporary patch for now that only did the HIGH$ check.

With two of the more critical parts out of the way, we now return to the disk I/O. I first tried each and every one of the commands and made a checklist of which ones malfunctioned and which ones didn't. The only two I found that failed were printing to the RS-232 and doing any disk file work in which I did not specify the filespec, but let VC look for me. Well, I didn't worry about the RS-232 problem, because it used ROM calls to the new Mod 3 system ROM, which doesn't even exist on the Mod 1 (see what I was talking about earlier?). Besides, anyone using LDOS who needs to print to the RS-232 has probably done a SET *PR TO *CL anyway, so why repeat the effort?

The file work then became the problem. At this point, DEBUG to the rescue! DEBUG is a vastly underrated utility which can be worth it's weight in gold. If you don't know how to use it, LEARN!! I can't stress enough how much easier it will make your work. The first thing to do is to set a breakpoint in your program. The question is, where to set it? Here comes another little hint: VC and many other programs employ a clever little trick - a lookup table for the commands. You can find in your HEX/ASCII dump a bunch of letters, each separated by 2 bytes, and these letters correspond to the VC commands. What they do is (1) point to the beginning of the table after getting the command, (2) compare the command to the character in the table it's pointed to, (3) if there's a match, put the NEXT TWO bytes of the table into HL, and JP (HL) or, (4) if NOT a match, move ahead THREE bytes in the table to the next command and repeat until you find a match or hit the end of the table. Once I found where /L (for file loading) jumped to, I set a breakpoint there, and started running. As soon as I typed /L, I dropped back to DEBUG and single-stepped my way along. This part can be rather tedious, but rewarding when you finally get results. As I went along, I made written notes so that when the program finally DID crash, I could restart and set my breakpoint farther along so I didn't have to repeat all the same single-stepping that I already KNEW was correct.

When I got to the subroutine that crashed the system, out came the disassembler again. I disassembled the subroutine's general memory area (just how much YOU disassemble will depend on the program, your mood, and how much paper you're willing to use up), and any other subroutines that were called by it. That done, I had to run back to the cross-ref to find out if any other portions of the program used parts of that same subroutine. Unfortunately, they did. I looked for the entry points, and made notes that all my changes had to stay before those parts of the subroutine so as not to mess up those other entry points.

VC's use of the RAMDIR call presented a problem, because LDOS uses @FNAME, which returns whatever is in a particular directory slot, whether it is an active slot or not. TRSDOS's RAMDIR returns 00 bytes if the directory slot is unused. Now, I had to find a way to tell if the directory slot was used. Also, if I had a double-sided disk or a hard disk, I had to be sure that I checked all possible directory slots. The answer was simple - look at the HIT sector in the directory track, and only read directory slots that had non-zero bytes in the matching HIT position!

That, however, created another problem - where to drop the HIT into memory for fear of destroying a table already in memory? Well, VC HAD to have a disk I/O buffer of it's own, so why not use that? After all, nothing would go there unless I asked for a file, and once I did, I wouldn't need the HIT anymore, anyway. It was decided. A quick glance back at the cross-ref told me where a file open/close/read operation was, and I knew that the last LD HL,xxxx command before that was to point to the I/O buffer. Buffer found, back to work.

VC uses a CALL 028DH to see if the BREAK key was hit during scan, so I had to mimic it by looking at bit 3 of 3840H in the keyboard memory. For some odd reason, VC chose to save ALL registers when only the AF registers were changed, so I cut out the extra PUSHes and POPs, thereby saving a few bytes.

Once VC has a drive number to search, I had to see if the drive was in the system, so I needed a call to @CKDRV. This would be one of the addresses to change between Mod 1 and 3, so it would later be modified in the HIGH$ testing routine. One other point was that TRSDOS and LDOS use the B & C registers in opposites on @CKDRV, DIRCYL, and @FNAME, so I had to include a a little routine of:

```
         LD A,C
         LD C,B
         LD B,A
         RET
```

to do the rotations.

Assuming the drive was in the system (if not, I skipped to the next drive or fell back out if all drives had been tested), I now wanted the HIT sector. A call to DIRCYL to find the directory, and RDSSEC to read system sector #1 (the HIT sector) into the VC disk buffer, and that step was over. Now, I entered a routine to look through that sector for a non-zero byte. Once found, a call to @FNAME to get the filespec. Only one more stumbling block - what if it was a directory EXTENSION (FXDE) rather than the PRIMARY entry (FPDE)? Well, if it's an FXDE, @FNAME thoughtfully drops back 00 bytes, so that became the last test to make. Actually, the second-last test. We finally had to test to see if the files had the /VC extension (or /DIF or /PRF, depending upon where the call originated) and pass it back for approval if it DID match, but that step was identical to the original routine already in the program, just re-located. If I didn't like the name that was chosen, I returned to my routine (with the needed registers saved, of course), and continued until all 256 directory entries had been checked, and then moved on to the next drive for a repeat of all the little fun and games above.

A word of caution... if you don't know what a certain piece of code does, LEAVE IT IN!! I found a flag being tested that I didn't understand, so I took it out. Bad move! It turned out that it was used in another part of the program, and the program hiccuped when I took it out. Sometimes you CAN take that kind of stuff out, but be careful! Anyhow, a little more testing, and the results were successful. Now, back to the HIGH$ routine to add the changes to alter the DOS vectors that varied between Model 1 and 3.

Almost done except for the /X- screen-refresh. This one was more trial-and-error than anything else. I found a similar lookup table with all the characters that could legally follow the /X, so I looked at the 2 bytes after the dash... they pointed directly to a RETurn instruction. Now, all I had to do was find the point at which the screen was refreshed normally, and put THAT address in its place. This turned out to be easier than I planned. Running back to DEBUG, I re-loaded and single-stepped from the beginning of the program, past the HIGH$ check and the memory-clearing routine, until I got to the point where the screen was first displayed. That found, I patched that address into the /X- slot of the table for testing. This was simple enough to do, since all it needed was to load some data onto the sheet, issue the /X- command, and check for any change in the data. No change meant I had the right spot (Or at least A right spot!).

Finally, a little alteration of the version display to note that it had been patched, and I was ready for a final complete assembly of the changes to get all the addresses and bytes to put into PATCH format. A quick run through LSCRIPT to create the /FIX file and I was done.

With all this work out of the way, it was immensely easier to make the patches for the newer version of VC (version VC-160Y0-T83 vs. the older VC-150Y0-T8). First, I did a HEX/ASCII dump of the new file and just looked for the movements of code. Fortunately, a few bytes of code were added near the beginning which moved everything down a few bytes, but none of the critical areas seemed to be altered. A re-assembly of the old code at a new origin, and everything worked (Ah, if only EVERYTHING were that simple...).

When you're done, be absolutely certain that you make notes on what features you've changed, and any important features you might have added to the program. After all, we all know how important documentation can be.

That's how it was done! I realize that not everything I did will apply to every situation or program, but the techniques here can be adapted for many uses. If you're just starting to learn machine language, some of the above may have been over your head. If you're an old pro, it may have bored you. In either case, I hope I've passed on some little tricks which you may not have known that will make life easier when it comes time to disassemble a program for corrections, or just "souping-up". By the way, you'll find below a reprint of the patches for VC version 150Y0 AND the patches for version 160Y0 to run on LDOS 5.1.2. If you want to move it to 5.1.3, just change the "21 90 42" in the X'5586' (or X'5591') line into "21 09 42" to move the @CKDRV call.

```
.Patches to Model 3 Enhanced VisiCalc to run on
.LDOS 5.1.2. By Ray Pelzer, partly based on the
.Model 1/3 patches (c) 1981 by Logical Systems, Inc.
.
.These patches will make VisiCalc 3 run on a Model 1 or 3.
.*NOTE* FOR VISICALC VERSION # VC-150Y0-T83 *ONLY*!
Patch in 002B kybd scan for typeahead, etc.
X'557F'=CD 2B 00
.
.All the stuff for proper HIGH$ and Mod 1/3 conversion.
X'521A'=CD 86 55
X'52A6'=CD 86 55
.Note:  for LDOS 5.1.3, change 90 to 09 in the line below.
X'5586'=3A 25 01 FE 49 3E 00 2A 49 40 C0 21 90 42 22 A8 55
X'5597'=21 64 4B 22 4A A7 21 93 42 22 69 A7 2A 11 44 C9
.
.Part of the @CKDRV call to see if a drive is active.
X'55A7'=CD B8 44 C8 CD 51 A4 E1 C3 3F A7
.
.Add the ever-popular "/X-" screen-refresh command.
X'7062'=5E 52
.
```

```
.Change the /V version display
X'9235'=28 63 29 20 31 39 37 39 2C 38 31 20 56 69 73 69
X'9245'=43 6F 72 70 20 56 43 2D 31 35 30 59 30 2D 54 38
X'9255'=33 20 20 4C 44 4F 53 20 62 79 20 52 61 79 20 50
X'9265'=65 6C 7A 65 72
.
.Now, make the changes to scan filespecs properly in LDOS
.When using a /S command withOUT an explicit filespec.
X'A43D'=F5 3A 40 38 E6 04 28 06 CD 2D A4 F1 37 C9 CD 35 A4
X'A44E'=F1 B7 C9 79 F5 78 4F F1 47 C9
X'A720'=22 88 B5 CD 35 A4 3A 8F B5 FE FF 47 20 02 06 00
X'A730'=CD 43 A7 D0 CD 3D A4 38 06 04 78 FE 08 38 F1 3E 03
X'A741'=37 C9 CD 51 A4 CD A7 55 CD 65 4B 1E 01 21 D6 B5
X'A751'=CD 45 4B CD 51 A4 20 55 21 D6 B5 7E E5 B7 C5
X'A760'=28 3A 11 D6 B6 CD 51 A4 CD BB 44 EB 7E B7 28 3E
X'A770'=7E FE 3A 28 39 FE 2F 23 20 F6 ED 5B 8A B5
X'A77E'=06 03 1A FE 20 20 05 7E FE 3A 18 07 BE 20 21 13 23
X'A78F'=10 EF 28 0C 3A 91 B5 B7 28 03 AF 18 02 3E 01 B7
X'A79F'=20 0B CD 17 A8 38 03 C1 E1 C9 CD D8 A7 18 03
X'A7AE'=CD C8 A7 C1 0C E1 23 CA 3F A7 18 A2
.END OF PATCH
-----------------------------------------------------------------

.Patches to Model 3 Enhanced VisiCalc to run on LDOS
.These patches will make VisiCalc 3 run on a Model 1 or 3.
.**NOTE** THESE PATCHES FOR VISICALC # VC-160Y0-T83 ONLY**
.Patch in 002B kybd scan for typeahead, etc.
X'558A'=CD 28 00

.All the stuff for proper HIGH$ and Mod 1/3 conversion.
X'521A'=CD 91 55
X'52B1'=CD 91 55
.Note: For LDOS 5.1.3, change 90 to 09 in the line below.
X'5591'=3A 25 01 FE 49 3E 00 2A 49 40 C0 21 90 42 22 B3 55
X'55A2'=21 64 4B 22 6B A7 21 93 42 22 8A A7 2A 11 44 C9
.
.Part of the @CKDRV call to see if a drive is active.
X'55B2'=CD B8 44 C8 CD 6F A4 E1 C3 60 A7
.
.add the ever-popular "/X-" screen-refresh command.
X'7075'=69 52
.
.Change the /V version display
X'924E'=28 63 29 20 31 39 37 39 2C 38 31 20 56 69 73 69
X'925E'=43 6F 72 70 20 56 43 2D 31 36 30 59 30 2D 54 38
X'926E'=33 20 20 4C 44 4F 53 20 62 79 20 52 61 79 20 50
X'927E'=65 6C 7A 65 72
.
.Now, make the changes to scan filespecs properly in LDOS
.When using a /S command withOUT an explicit filespec.
X'A45B'=F5 3A 40 38 E6 04 28 06 CD 4B A4 F1 37 C9 CD 53 A4
X'A46C'=F1 B7 C9 79 48 47 C9
X'A741'=22 A9 B5 CD 53 A4 3A B0 B5 FE FF 47 20 02 06 00
X'A751'=CD 64 A7 D0 CD 5B A4 38 06 04 78 FE 08 38 F1 3E 03
X'A762'=37 C9 CD 6F A4 CD B2 55 CD 65 4B 1E 01 21 F7 B5
X'A772'=CD 45 4B CD 6F A4 20 55 21 F7 B5 7E E5 B7 C5
X'A781'=28 3A 11 F7 B6 CD 6F A4 CD BB 44 EB 7E B7 28 3E
X'A791'=7E FE 3A 28 39 FE 2F 23 20 F6 ED 5B AB B5
X'A79F'=06 03 1A FE 20 20 05 7E FE 3A 18 07 BE 20 21 13 23
X'A7B0'=10 EF 28 0C 3A B2 B5 B7 28 03 AF 18 02 3E 01 87
X'A7C0'=20 0B CD 38 A8 38 03 C1 E1 C9 CD F9 A7 18 03
X'A7CF'=CD E9 A7 C1 0C E1 23 CA 60 A7 18 A2
```

LDOS DISK DRIVE CONTROL LINKAGES

by Bob Bowker <70250,306>

One of the main reasons for the portability of so much of the CP/M software on the market today is their use of a system vector address which is common to all versions of CP/M or "CP/M look-alikes". System routines may vary in length or actual memory location from one version to the next - in fact, they may be totally different from each other - yet both systems will be able to handle the same software because of the single system vector.

Here's how it works in CP/M. To display a byte on the CRT, for example, the user loads the byte into the DE register pair, loads function number 02 (Write to console) into the C register, and CALLs 0005H. To send that same character to the line printer, function number 05 (Write list) is loaded into the C register, and a CALL 0005H is executed. And so on.....CP/M 2.2 and MP/M 1.0 define 36 different functions, including disk I/O, which are accessed in this way.

Next to program portability, the biggest advantage to using a single system entry point is device independence. The system vector at 0005H sends the caller into upper memory where the individual device drivers and interface code are located. Whether the printer attached to the system is a high speed dot matrix on a parallel port, or a teletype on a serial port, makes no difference to the caller's program - the byte to be printed is passed in the same register pair in both cases, and the C register contains a 05 in both cases. Similarly, neither type of disk drives, nor the protocol of the terminal, is of any importance to the caller's program - in every case, the needs of the individual peripherals are met by the system drivers, freeing the caller's program of those responsibilities.

The actual routines to which the user's CALL is vectored are at different addresses depending on system version numbers, what peripherals are attached, memory size, etc., but the user does not have to know the exact location - the system maintains 0005H as a vector to the entry point. LDOS has taken the first steps in this direction by applying these principles to 16 disk I/O functions (see the Manual, page 6-15). All 16 functions are accessible by passing a function number in a CALL to the driver program for that particular disk drive, regardless of whether the drive is a floppy or hard disk, 5" or 8", etc.

Some of the 16 functions are also available to assembly language programmers as "primitives" - that is, via CALLs to unique RAM addresses. However, each of these primitives does little more than load the function number in the B register, load the appropriate driver program address in the IY register pair, and JUMP there. One very important point: none of these primitives is a "fixed" address as of 5.1.2 - that is, they may move in future releases. The location of the Drive Code Table, however, is a fixed address - DCT$ = 4700H. The primitives are listed in the following table, along with their linkage function numbers:

ADDRESS	CODE	NAME	OPERATION
	00H	NOP	tests if drive is assigned in DCT
4754H	01H	SELECT	new drive/return status
	02H	INIT	set cyl 0, RESTOR, and set side 0
	03H	RESET	reset FDC
	04H	RESTOR	move head to cyl 0
	05H	STEPIN	move head "in" one cylinder
475EH	06H	SEEK	seek a specified cylinder
44B8H	07H	TSTBSY	test drive for busy
	08H	RDHDR	read one sector header info
4777H	09H	RDSEC	read one specified sector into mem
4772H	0AH	VERSEC	verify that one sector exists
	0BH	RDCYL	read one whole cylinder

Let's look at a couple of examples. The following code will read a sector from diskette into a buffer in RAM:

```
00100 READ1     LD     A,(DRIVE)      ;get drive number
00110           LD     C,A            ;into proper register
00120           LD     A,(CYLNDR)     ;get cylinder number
00130           LD     D,A            ;into proper register
00140           LD     A,(SECTOR)     ;get sector number
00150           LD     E,A            ;into proper register
00160           LD     HL,BUFFER      ;point to storage area
00170           CALL   4777H          ;and read it in.
00180           RET
```

This routine assumes only that (DRIVE), (CYLNDR), and (SECTOR) are maintained by the program which CALLs READ1. The following code will do the job, too:

```
00200 READ2     LD     B,09H          ;linkage code for RDSEC
00210           LD     A,(DRIVE)      ;get drive number
00220           LD     C,A            ;into proper register
00230           LD     A,(CYLNDR)     ;get cylinder number
00240           LD     D,A            ;into proper register
00250           LD     A,(SECTOR)     ;get sector number
00260           LD     E,A            ;into proper register
00270           LD     HL,BUFFER      ;point to storage area
00280           LD     IY,4700H       ;point to drive's DCT
00290           JP     (IY)           ;and read sector in.
```

Change the number loaded into the B register in line 200 to an 0DH, and this same code will WRITE that sector; make it an 0AH, and this same code will VERIFY that sector; and so on through all 16 available functions. Note that the RETurn at the end of the LDOS routines will send the program back to the code which called READ2. The DCT is a table residing from 4700H to 474FH; each of the 8 possible drives is allocated 10 bytes in which the information about that drive is stored (see the Manual, page 6-11, for the details). The first three bytes of each drive's DCT form a JUMP to the driver which controls it; if the drive isn't active, a RET is in the 1st byte instead of the JP (a C9H replaces the C3H - the driver address in relative bytes 1 and 2 is always in place, whether the drive is active or not).

In the above example, relative drive 0 is assumed; the vector to its driver program is located at the start of the table, at 4700H. This method of disk I/O is not necessarily "better" than using the primitives, but in many long or complex programs it could save a lot of bytes. Consider the following subroutine:

```
00300 SUB1      LD     A,(DRIVE)      ;get drive number
00310           LD     C,A            ;into proper register
00320           LD     A,(CYLNDR)     ;get cylinder number
00330           LD     D,A            ;into proper register
00340           LD     A,(SECTOR)     ;get sector number
00350           LD     E,A            ;into proper register
00360           CALL   GETDCT         ;load DCT address in IY
00370           JP     (IY)           ;and go do the job.
```

This code serves as a general disk I/O routine; it will perform the function defined by the contents of the B register when it's CALLed. For example:

```
00400 PROG      LD     B,09H          ;linkage code for RDSEC
00410           LD     HL,BUFFER      ;point to storage
00420           CALL   SUB1           ;   and go do it
00430 BACK      ....                  ;continue on.
```

Here again, the RETurn at the end of the LDOS routines will send the program BACK in your code. With regard to the CALL GETDCT in line 360, you have two choices: if GETDCT is EQUated to 478FH, LDOS will do the work for you - load the DCT address (for the drive specified in the C register) into the IY register pair; or, since 478FH is not a "fixed" address as of LDOS 5.1.2 (it may move in a future release), you can include your own GETUCT subroutine in your program:

```
00500 GETDCT    LD      IY,4700H      ;beginning of DCT$
00510           LD      A,C           ;check drive number
00520           OR      A             ;...0s it 0?
00530           RET     Z             ;done if it is
00540           PUSH    BC            ;save register for now
00550           LD      B,A           ;put drive number into B
00560           XOR     A             ;zero the A register
00570 LOOP      ADD     A,0AH         ;bump the offset
00580           DJNZ    LOOP          ;.. "B" times
00590           LD      C,A           ;BC==> total offset
00600           ADD     IY,BC         ;IY==> drive's DCT address
00610           POP     BC            ;restore BC contents
00620           RET                   ;and go back.
```

This routine sets IY=4700H right at the start; if the drive number is 0, it's all over and a RETurn is executed. If it's not, we must calculate the offset from 4700H to the start of the DCT for the requested drive.

The DJNZ instruction bumps the A register offset counter by l0 for each relative drive position up to the requested drive number. By the time it falls through to line 590, the A register will contain 10 times the relative drive number (l0, 20, ..., 70 in decimal, or 0AH, 14H, ..., 46H) which, when added to the start of the table in IY (4700H) yields the UCT address of the desired drive.

On return from the CALL SUB1 in line 420, all registers will reflect the same information as they would have had the primitive been called (error codes, etc.). The use of these linkages is not necessary for every program nor for every application; the main advantages lie in the time and byte savings their use offers, and the "version independence" they afford you. Certainly no time would be saved by including the above subroutines for only one or two uses - in that case, the primitives will serve nicely. On the other hand, if your program uses the LDOS routines which have defined linkages very often, they would be worth trying.

LSCRIPT patches to add versatility

by Scott Loomer

The Quarterly usually does not accept articles about modifying the LDOS Libraries or Utilities. This article is unusual in that it describes how to modify the "standard" LDOS version of LSCRIPT. It's usefullness and flexability enhance the standard LSCRIPT version of Scripsit enough that I (the Editor) felt that it could be included.

This article is to describe the variety of patches, filters and techniques that I use to make Scripsit a very powerful word processor. These patches and techniques will only work under LDOS, but then that's why you are reading this article. Some of the techniques are suitable only for the Epson MX-80 printer w/Graftrax or printers that use similar control schemes.

Patches

First and foremost is, of course, the LSCRIPT patch provided with LDOS. This patch should be applied per instructions to a virgin copy of Model I Scripsit. LSCRIPT provides the following enhancements:

- use of LDOS keyboard driver allowing full ASCII characters and use of all minidos functions
- re-entrant capability to Scripsit with Scripsit *
- imbedded printer control codes at text boundaries
- keyboard insertion of text at text boundaries
- directories and partial directories are available within Scripsit
- consistent use of <CLEAR> key as control key
- default file extensions
- and lastly, Scripsit no longer reboots your system on exit

The LSCRIPT patch is described in greater detail in the LDOS documentation. I will assume that you are using this patch as a basic starting point.

The remaining patches that I use are provided as figure 1 in this article. Some of the six patches have appeared before, but they are presented here to bring them all together. I wrote some of the patches while others are modifications of patches done originally by Earle Robinson and Les Mikesell who have permitted their use here. Special thanks also go to Roy Soltoff who supplied the locations of some memory blocks in Scripsit available for use by patches 4 & 6 (and I used darned near all the bytes that were available). The comments in the patch file indicate the authors. Each of the six patches is independent from the others; you may use only those that appeal to you and delete (or comment out) the rest. All of the patches are direct (D) patches which will not extend the length of Scripsit. Some of the patches are for the Model III only and the patch lines are commented out so that they won't affect Mod 1's. Model III users will need to remove the periods at the beginning of those patch lines they wish to use. Let's examine the patches in order:

1. Patch 1 - This patch enables the following key functions:
   <CLEAR><Up Arrow> - move to start of text
   <CLEAR><Down Arrow> - move to end of text
   <CLEAR><Right Arrow> - tab to next tab stop
   <CLEAR><Left Arrow> - move to start of line

This is a logical extension of the use of the <CLEAR> key as the control key for Scripsit.

2. Patch 2 - If your printer needs a line feed instead of a carriage return to advance to the next line, apply this patch.

3. Patch 3 - This patch is purely cosmetic and consists of some recommended patches to symbols used in Scripsit. Be sure to note that those that are Model III only are commented out. You may substitute your own characters by patching the same addresses with your choice.

4. Patch 4 - Scripsit and PR/FLT don't get along well as you may have discovered by now. The problem is that both are trying to format the printer output. If you normally run with PR/FLT installed this patch is for you. On entry to Scripsit, the patched code checks DFLAG$ to see whether or not PR/FLT is active. If PR/FLT is active, the current parameters for left margin, characters per line and lines per page are located. Since PR/FLT could be anywhere in memory, it has to be found first. A pointer to PR/FLT is maintained at memory location 4DFD. This pointer is required by LDOS to allow it to reuse the memory if PR/FLT is re-installed in the system. The pointer points to the beginning of PR/FLT where the parameter block is located. The patch picks up the values for the parameters mentioned above and stores them internally. The parameters in the filter are then set to no left margin, 256 character line length and 66 lines per page. Scripsit is then entered. The effect of the above is that while in Scripsit, the formatting is controlled by Scripsit's format commands. If you specify a left margin of 0, the text will not be indented even if you had set the margin parameter in PR/FLT. Likewise, you'll get accurate formfeeds. When you exit Scripsit with the END command, the above process is reversed and the PR/FLT parameters are restored to their original settings. As mentioned earlier, the patch checks to see if PR/FLT is active and can therefore be used with or without PR/FLT.

5. Patch 5 - This patch is a reworked version of SFILE by Les Mikesell. I have found SFILE to be the most useful enhancement to Scripsit that I use, BUT it was originally written as an appendage to Scripsit which caused it to have two undesirable effects. First, at least on the Model III, it extended Scripsit by enough to cause it to use an extra granule (12k instead of 10.5k). Second, and more importantly, it defeated the re-entrant capability added by LSCRIPT. This was caused by the fact that the original SFILE loads into the text buffer area of Scripsit and then relocates itself to high memory. This obviously has the effect of clobbering a portion of the text buffer and so the re-entrant capability was disabled. Patch five is a direct patch version of SFILE that resides internally to Scripsit (does not extend it) and is therefore re-entrant. This new SFILE still uses high memory for a file buffer and file control block, but you'll have more buffer space available than you did before. The commands that SFILE adds are:

P,D - Print to the display. This command will "print" the formatted text to the display. Pressing keys 0-9 control the scrolling speed. Nine is the slowest and 0 the fastest. Pressing any other key will stop the display; pressing another key will restart it. To abort the display early, press <BREAK> twice. After the text has scrolled to the end, you can get your display back by pressing <BREAK>.

P,F - Print (formatted) to a file. This command will first clear the screen and then prompt for a "Filespec:". Enter the filespec that you want to save the text to. Note that during the entry of the filespec, you are not in the extended cursor mode. This means you can't backspace with out shifting out of ECM (see the KI/DVR documentation). The file will then be written formatted as it would have been sent to the printer.

6. Patch 6 - Corrects a call in LSCRIPT to the @CKDRV vector that was changed in Mod III LDOS 5.1.3. Can be patched on both Mod I & III with no ill effects.

7. Patch 7 - Actually, I lied earlier when I said that all of the patches were direct patches. Patch seven is an X patch for the Mod III only. It ensures that if you are using the Model III special paragraph character that you get it and not the alternate katakana character when you enter Scripsit. This patch extends Scripsit by a few bytes but it doesn't cause it to use another granule.

Techniques

Depending on your printer, there are other capabilities readily available in LDOS that will enhance your use of Scripsit. As mentioned under patches, I routinely use PR/FLT. If your printer will accept a hard formfeed, be sure to use that parameter when you install PR/FLT. Since Minidos now works from within Scripsit, you can get a fast top of form with <CLEAR><SHIFT><T>. Note that some printers may take a hard form feed, but not advance the paper any faster that the normal series of line feeds.

A second parameter of PRIFLT that can be very useful is XLATE. If your printer is capable of multiple print styles and/or pitches, look at the command syntax used to shift modes. If you are in luck, it will be an escape (27 or 11H) letter sequence. This is the case on the MX-80 with Graftrax. Now, Scripsit doesn't allow imbedded control codes (except at text boundaries with LSCRIPT) but you have some characters available with KI/DVR that are expendable. The best one is the delete character (127 or 7FH) which is displayed as a plus/minus (Mod III) or a block (Mod I). Many printers won't respond to this character and, anyway, why do you want to delete a character that has already been sent to the printer? Instead, XLATE=x'7F1B' which will translate the delete character into the escape character on output to the printer. Thus the following characters anywhere in Scripsit will cause my MX-80 to shift into the emphasized mode: +/-E (of course you need to create the plus/minus character with <CLEAR><SHIFT><ENTER>).

Using this method, I can change print modes anywhere in the text. There is one slight problem; if you are using right justified text, Scripsit will count your printer control characters even though they don't get printed, and throw the right margin off.

The final indispensable feature of LDOS when you are using Scripsit is the spooler. I am always amazed to find that people do not routinely use the spooler. Scripsit is the perfect application for it since word processing (entering and editing text) is a very low demand task for the CPU. This means it spends most of its time waiting for you to press a key (no matter how fast you type). Put it to work. Install the spooler and have it print one document while you edit the next. You can tailor the size of the spooler's memory and disk usage to meet your needs. About 3k of spooler space is required for one page of text.

Filters

There are several possibilities for using filters to enhance Scripsit. The use of PR/FLT had already been described. If you want to translate further characters (or not use PR/FLT), a simple one byte translate filter can be written using the example filters in the LDOS documentation as a guide. Some candidates for translation are using some of the less commonly used ASCII characters as printer control codes or to create a non-expandable blank. What is a non-expandable blank? Well, how often have you prepared a document in Scripsit that had a proper name such as John Q. Doe. Murphy's law guarantees that the name will be broken at the end of a line and you'll end up with John
Q. Doe which doesn't look very nice. Try this instead; use a translate filter to convert a character such as @ (anything you don't use much) to a space. In your document you type John@Q.@Doe. Scripsit will treat this as a single word when formatting, but it will be printed correctly. The last filter I routinely use is an underline filter. This filter is available as part of the GRASP package from MISOSYS. It will allow you to choose an underline delimiting character (perhaps a tilde) and will automatically underline any text bracketed by this character. Your printer must be capable of non-destructive backspacing. I am somewhat biased towards the use of this filter as I wrote it.

That about exhausts what I have to say about enhancing Scripsit. I am quite pleased with the final product and figure that it has saved me at least $100 as I see no major advantage in switching to SuperScripsit. Enjoy.

```
.LSCRIPTX Patch - Enhancements to LSCRIPT
.The following patches are to be used after applying the
.LSCRIPT patch provided by LSI. Be sure to enable those
.lines that apply to your machine if using a Mod III.
.Use this patch only with LDOS 5.1 or later as indicated.
.
.Patch 1 - enable <CLEAR><Up, Down, Left and Right Arrow>
. Originally known as LSCR2/FIX by Earle Robinson
. Later modified as LSCR3/FIX by Scott Loomer to add the
. right and left arrow functions.
. This patch creates the following capabilities:
.    <CLEAR><UP Arrow> - moves to start of text
.    <CLEAR><Down Arrow> - moves to end of text
.    <CLEAR><Left Arrow> - moves to start of line
.    <CLEAR><Right Arrow> - tabs to next tab stop
.  This is consistent use of the <CLEAR> key as the Scripsit
.  control key
D0E,6A=CD 0E 52 1E 17
D00,12=21 4D 60 57 FE 88 20 04 LE 1A 18 15 FE 82 20 04
D00,22=1E 92 18 0D FE 84 20 04 LE 09 18 05 FE 81 C0 1E
D00,32=91 3A 40 38 CB 4F 7A C8 7B C9
.End of patch
```

```
.Patch 2 - change line feeds to carriage returns
.  This patch will cause carriage returns instead of line
.  feeds to be sent at end of lines as required by some
.  printers. Patch provided by Roy Soltoff.
D1F,F1=0D
D20,05=0D
.End of Patch
.
.Patch 3 - change certain symbols
.  Each of the following lines change one of the symbols in
.  Scripsit. Certain of the replacement characters will not be
.  displayed correctly on Mod I's due to different character
.  set. Locations determined by Earle Robinson & Scott Loomer.
.Change block markers to [ and ] respectively on Mod 3.
.D28,0A=5B
.D28,0C=5D
.Change paragraph symbol to paragraph character on Mod 3 only
.D28,0E=F1
Change cursor to underline
D00,DD=5F
.Change insert character to smaller graphic block
D04,8F=B0
.Change bottom border character to underline
D19,A1=5F
.Change end of line marker to small graphic block
D28,08=84
.Change insert block character to smaller graphic block
D28,D3=5F
.End of Patch
.
.Patch 4 - resolve conflicts between Scripsit & PR/FLT
.  The following patch allows Scripsit to peacefully coexist
.  with PR/FLT. The patch will zero the PR/FLT parameters on
.  entry to Scripsit and restore them on exit. This will only
.  work for LDOS 5.1.2 and later. This patch was written by
.  Scott Loomer.
D00,48=C3 F5 63
D12,3D=3A 25 01 FE 49 28 07 21 1F 44 3E
D12,4C=42 18 05 21 89 42 3E 43 32 32 64 22 A1 62 7E CB
D12,5C=5F 28 23 DD 2A F6 4D DD 7E 19 32 39 64 DD 36 19
D12,6C=00 DD 7E 1A 32 3A 64 DD 36 1A 00 3A 2A 40 32 3B
D12,7C=64 3E 00 32 2A 40 C3 69 63 00 00 00
D13,E4=C3 A0 62
D10,E4=3A 00 00 CB 5F 28 16 DD 2A F6 4D 3A 39 64 DD 77
D10,F4=19 3A 3A 64 DD 77 1A 3A 3B 64 32 2A 40 C3 DD 63
.End of Patch
.
Patch 5 - adds print to file & display functions
.  This patch was originally developed as SFILE by Les
.  Mikesell. It was modified by Scott Loomer to make it reside
.  internally to Scripsit. The re-entrant capability
.  (Scripsit *) added by LScript now works.
.  NOTE: It is absolutely necessary after applying this patch
.  to use CMDFILE to change the transfer address of Scripsit
.  to 6238H after applying this patch
D06,21=2B 20 46 69 6C 65 20 26 20 44 69 73 70 6C 61 79
D06,48="Version 1.5"
D00,0B=DC 60
D00,6B=21
D1E,DD=C3 70 62
D19,73=98 62
D06,0E=40
```

```
D14,38=92 62
D10,7C=E5 3A 25 01 2A 11 44 FE 49 28 03 2A 49 40 01 20
D10,8C=01 AF ED 42 22 68 52 23 22 F1 60 22 0C 61 22 26
D10,9C=61 22 46 61 22 4E 61 3E 08 77 01 20 00 09 22 1E
D10,AC=61 22 30 61 E1 C3 00 52 FD CB 0E 5E C4 12 61 FD
D10,BC=CB 0E 66 C4 C9 01 CD C9 70 FD CB 0E 5E C4 4A 61
D10,CC=FD CB 0E 66 C4 49 00 C3 64 70 49 53 50 46 44 00
D10,DC=3E 08 32 00 00 C3 D5 65
D0F,18=FD CB 0E 66 20 09 FD CB 0E 5E 20 08 C3 3B 00 F5
D0F,28=CD 57 61 F1 11 00 00 FE 0A 20 02 3E 0D FD CB 0E
D0F,38=5E C4 1B 00
D0F,40=C8 F6 C0 CD 09 44 CD 28 44 3E 08 32 00 00 CD 49
D0f,50=00 C9 D5 E5 C5 21 7F 61 CD 67 44 06 1F 21 00 00
D0F,60=CD 40 00 38 1E 11 00 00 CD 1C 44 20 0E 06 00 21
D0F,70=00 00 CD 20 44 20 04 C1 E1 D1 C9 F6 C0 CD 09 44
D0F,80=CD 49 00 3E 08 32 00 00 18 ED D5 E5 C5 11 00 00
D0F,90=CD 28 44 20 E6 18 EC F5 CD 2B 00 B7 28 0D FE 30
D0F,A0=38 15 FE 3A 30 11 D6 30 32 6E 61 C5 01 FF 04 CD
D0F,B0=60 00 C1 F1 C3 33 00 CD 2B 00 B7 20 F6 18 F8 1C
D0F,C0=1F 46 69 6C 65 73 70 65 63 3A 0E 03
.End of Patch

.Patch 6 - fixes LScript for 5.1.3 on the Mod 3
.  There is a call to @CKDRV in the LScript patch and that's
.  the vector that was changed in 5.1.3. Without this patch
.  you will be unable to get a directory in LScript.
.  This patch is only required on the Mod 3.
.D11,CF=09
.End of Patch
.
Patch 7 - force the special character set on Mod III's
.  This patch will ensure that if you are using the paragraph
.  symbol on the Mod III (see patch 3) that you will get it
.  and not the Katakana character. Patch by Scott Loomer.
.  Note: this has to be an X patch as it works externally to
.  Scripsit.
.X'4210'=38
.End of Patch
```

If you have any questions concerning this article, please refer them to:

Scott A. Loomer
315 Palomino Lane
Madison, WI 53705
608-233-7739 or MicroNet [70075,1033]

Earl Terwilliger At Large

Earl has written an interesting article on short and simple Terminal programs for the TRS-80. In addition, he has used Roy's article on task processing to create a new "Alive" command program.

Have you ever seen a terminal program for the TRS80 as short as this ?

```
10 CLS:PRINT @ 200 "COM/BAS WRITTEN BY EARL TERWILLIGER"
20 OUT232,0:OUT233,85:OUT234,165
30 X=INP(234):IFX<128THEN40ELSE:X=INP(235):IFX=10THEN30ELSE:
   PRINTCHR$(X);:GOTO30
40 X$=INKEY$:IFX$=""THEN30ELSE:OUT235,ASC(X$):GOTO30:END
```

(CONTINUED ON PAGE 51)

BEFORE YOU GET TOO EXCITED ABOUT LOBO'S NEW COMPUTER, THERE'S SOMETHING YOU SHOULD KNOW.

There's plenty to be excited about in Lobo's new MAX-80,™ as you'll see in just a minute.

But first we want to warn you: you can't get one right away. Already, orders are coming in faster than we can build systems. However, if you can appreciate an incredible price/performance bargain, you'll agree the MAX-80 is well worth waiting for.

WHAT'S ALL THE EXCITEMENT ABOUT?

We're glad you asked. And the answer is pretty simple. Just look at this list of *standard* features:

- *5 MHz Z-80B processor.* That's 2½ times the speed of a TRS-80 Model III or Soft-Card/Apple!
- *64k RAM.* 128k is a low-cost option.
- *CP/M included.* A few more dollars get you LDOS, an incredibly powerful operating system that lets you run standard Radio Shack software.
- *Software-selectable 25 x 80, 16 x 64, and 16 x 32 screen formats.* For full compatibility with CP/M and TRS-80 applications.
- *All disk interfaces built in.* Plug in any combination of 5¼" floppies, 8" floppies, and Winchester disk.
- *Two RS-232 serial ports.* Ready to plug in modems, printers, or what-have-you.
- *Centronics-type parallel port.* For any printer using this standard interface.
- *Plus:* numeric keypad with 4 function keys, software definable text and graphics characters, built-in clock/calendar with battery backup, and buffered I/O expander port.

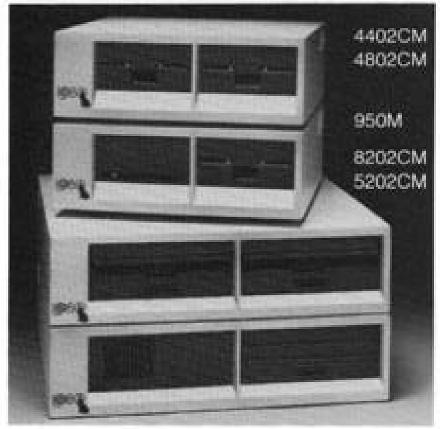
Now for the best part: *the factory-direct price for all this power is just $820—including shipping and Lobo's standard 1-year hardware warranty!*

WHAT TO DO NOW.

Call Lobo toll-free. Tell us what hardware and software configuration you're interested in (see below), and we'll give you an approximate shipping date. A $100 deposit will hold your place on the waiting list.

Then get a good book to help you pass the time.

MAX-80



CPU and Accessories

MAX-80 computer with 64k RAM and CP/M	**$ 820.00**
64k expansion RAM (installed)	**$ 95.00**
12" (diag.) high-resolution anti-glare green phosphor monitor	**$ 175.00**
LDOS operating system *instead of* CP/M	**$ 39.00**
LDOS operating system *in addition to* CP/M	**$ 69.00**

Dual 5¼" Floppy Disk Systems

4402CM single-sided, 40 track; 180 kB per diskette	**$ 690.00**
4802CM double-sided, 80 track; 720 kB per diskette	**$1,175.00**

Dual 8" Floppy Disk Systems

NOTE: Lobo CP/M permits reading and writing standard single-sided, single density CP/M disks with either of these systems.

8202CM single-sided, double density; 577 kB per diskette	**$1,185.00**
5202CM double-sided, double density; 1155 kB per diskette	**$1,485.00**

Winchester Disk Systems

950M 5¼" system: 4.8 MB hard disk plus 720 kB floppy	**$2,405.00**
950MX same as 950M above but no floppy drive	**$2,100.00**
1850M 8" system: 8.0 MB hard disk plus 1155 kB floppy	**$3,085.00**

The Lobo Warranty

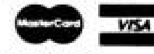
All Lobo hardware products carry a limited 1-year parts and labor warranty. Call or write for complete warranty statement.

© 1982 Lobo Drives International, Goleta, CA
CP/M trademark of Digital Research Corp.
TRS-80 trademark of Tandy Corporation.
SoftCard trademark of Microsoft Corporation.
Apple trademark of Apple Computer.
LDOS trademark of Logical Systems Incorporated.

ALL PRICES INCLUDE SHIPPING WITHIN THE U.S.A. California residents add 6% sales tax. Payable by credit card, check, or money order.

TOLL-FREE ORDER NUMBERS:

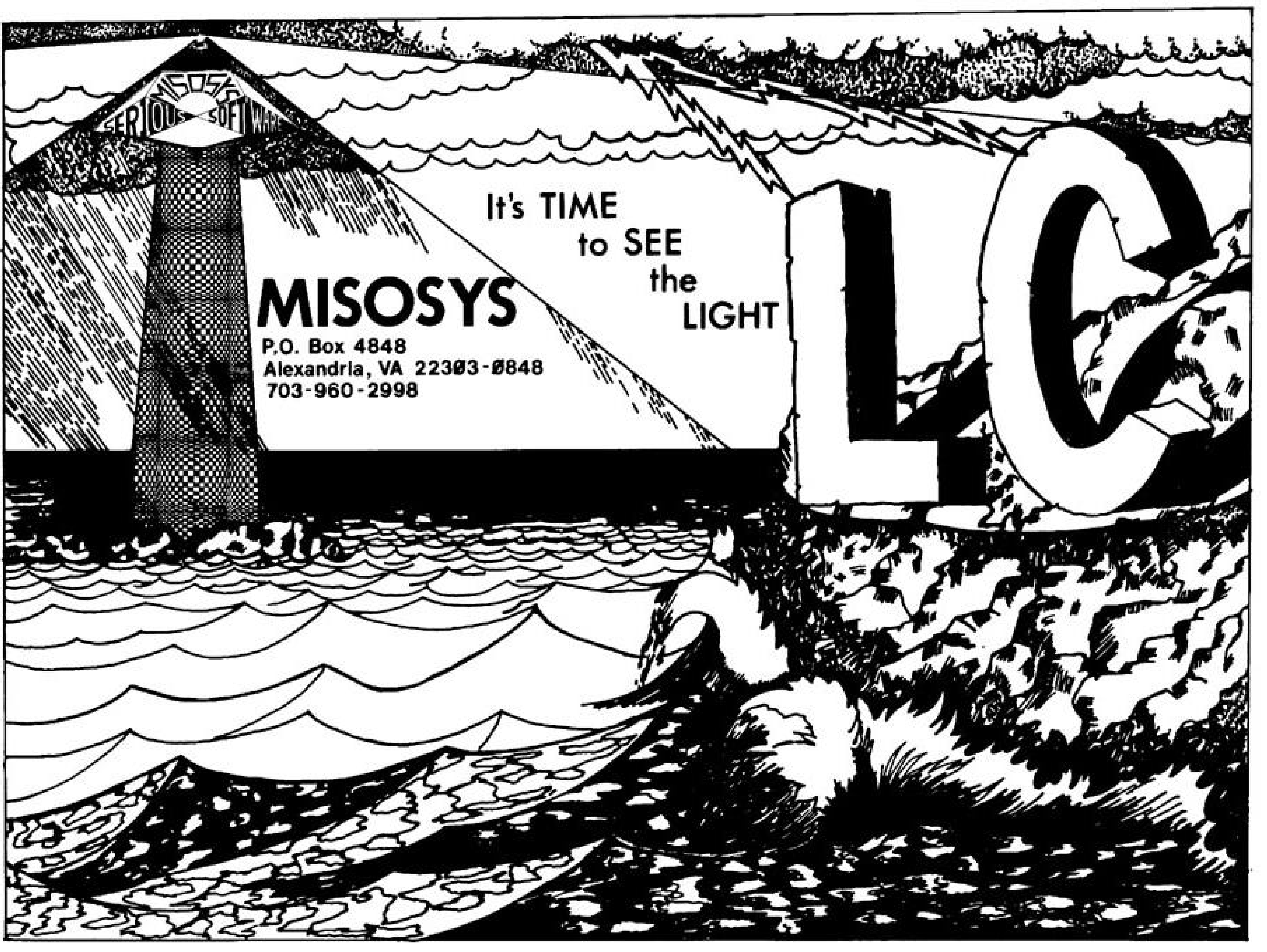
U.S. (except California):
800-235-1245



In California: **800-322-6103 or 800-322-6104** Hours: 7AM—5PM Pacific Time

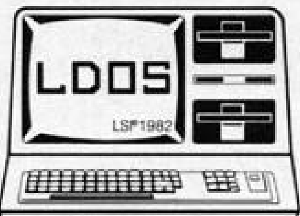
Lobo Drives International
Dept. LDS
358 S. Fairview Ave.
Goleta, CA 93117

Prices subject to change without notice.


SERIOUS SOFTWARE
It's TIME
to SEE
the
LIGHT
MISOSYS
P.O. Box 4848
Alexandria, VA 223Ø3-Ø848
703-960-2998
LC

LDOS MICRO REVIEW

Volume 1 No. 1 ★ SPECIAL EDITION ★★★ December 1, 1982



You'll think you've made the DOS strike of the decade when you turn your micro on to LDOS. You'll find a bonanza of features like full keyboard type-ahead; a true background spooler; file backup by date, class, and between different drive types; hard disk support; data transportability between Model I and III; and a complete communications utility including disk file send and receive. Support for Radio Shack's Doubler and selected others is also provided.

With our Job Control Language, you get true "hands off" running of your application programs — give a single command and then walk away. The 400 page manual includes examples of all commands and utilities. The Operator's Guide gives you step by step instructions on how to use LDOS with your applications. Stop running with only "half" a computer! Let LDOS provide the missing features to speed up and simplify your TRS-80 computer system! Visit a dealer or contact LSI for more information on the most popular sophisticated operating system for your TRS-80.

LDOS is available worldwide through thousands of dealers for just $129.

The BASIC Answer

The BASIC Answer is a BASIC text processing utility. It is designed to allow the BASIC programmer to build code in a structured manner. "Source" code is written with a word processor or text editor which allows the user to exploit the powerful editing and movement features characteristic to those types of editors. Source code can even be created by your own BASIC interpreter. **The BASIC Answer** is then used to process these files into normal interpretive BASIC code.

Free Yourself from Line Numbers

The BASIC Answer allows substitution of labels for **line numbers!** This means that your BASIC code now can read like a novel. Instead of the typically undescriptive "GOSUB 1000", a label such as "GOSUB @Search.Name" is used. Imagine yourself reading code filled with such descriptive branches and understanding it at a glance, even years later. This feature even allows totally relocatable BASIC routines without the renumbering problems.

TRS-80 is a trademark of Tandy Corporation. LDOS is available for the TRS-80 Model-I and Model-III. Prices and specifications subject to change without notice. LDOS and The BASIC ANSWER are products of Logical Systems, Inc.

A New Concept in Variable Usage

The BASIC Answer allows variable names to be as long as 14 characters and ALL 14 are significant. Imagine reading:

"IF ACCNT.OVERDUE #> 0 THEN GOSUB @PRINT.DUN"

rather than

"IFAO#>0THEN GOSUB52130"

Which would you rather read? It also introduces to BASIC the concept of Global and Local variables. This feature circumvents the tedious problem of variable tracking because a Local variable is only viable in its own subroutine!

NOW AVAILABLE
LDOS 5.1 Quick Reference Card
$5.95.

End the Multiple Machine Hassle

The BASIC Answer introduces the concept of **"Conditional Translation."** This feature allows the programmer to place different "machine dependent" code simultaneously into the same Source Code. **The BASIC Answer** can be "switched" when processing to ignore the unwanted or include extra code! No more multiple master programs to confuse maintenance. All the masters could now be rolled into the same program. Modify the one master and you've modified them all. Process the same code with different switches set, and get two or more versions from the same source.

The BASIC Answer combines the self-documenting power of COBOL with the relative ease of BASIC together with the power of a word processor.

The BASIC Answer is available for just $69.00.

11520 N. Port Washington Rd.
Mequon, WI 53092
(414) 241-3066

UTILITY DISKS from POWERSOFT

11500 STEMMONS FWY.
SUITE 125
DALLAS, TEXAS 75229

PowerSOFT's Utility Disks for LDOS saves time, money, and frustation. These fine utilities are written by Kim Watt, author of Super Utility Plus, and are in machine language. ALL drive configurations are supported, including single and double density, double sided, 5", 8", and all hard drives running under LDOS, including Radio Shack and Laredo.

---

* PMOD/CMD - A sophisticated Disk/File/Memory Utility.
* PCHECK/CMD- LDOS Directory Check Utility.
* PFIX/CMD - Repairs GAT Tor HIT table, BOOT sector, or files.
* PFIND/CMD - Finds strings, bytes, or words. Optional REPLACE.
* PCOMPARE/C- Compare a File/Sector to any other for difference
* PREFORM/CM- Reformat without Erase. Fixes CRC errors!
* PVU/CMD - Verifies disk for bad sectors.
* PERASE/CMD- Disk Bulk Eraser. (for 5" floppy use only)
* PCLEAR/CMD- Removes ALL traces of "KILLED" files.
* PSSTAT/CMD- Sector Staus. Finds what file is where.
* PMAP/CMD - Disk/File Mapper. Shows all files with extents.
* PMOVE/CMD - Super FAST multiple copy utility.
* PDIRT/CMD - Read a Mod III TRSDOS(tm) disk from LDOS.
* PASSGO/CMD- Remove passwords from a file or a whole disk.
* PUN/CMD - UN-Repair an LDOS disk. (Mod I only)
* PEX/CMD - A disk exerciser for head cleaning kits.
* PMX/CMD - Screen print graphics on an Epson printer.
* PHELP/CMD - A very complete HELP command for LDOS.
* PBOOT/CMD - Customize the way your LDOS boots up!
* PFILT/FLT - a USER definable printer filter.
* DVORAK/FLT- DVORAK/QWERTY keyboard filter.
* CODE/JCL - Keyboard encoding filter.(ENCRYPTES)
* DeCODE/JCL- Keyboard DECODING filter.(DECRYPTES)

COMPLETE WITH MATCHING DOC FOR YOUR LDOS BINDER

	#2	#3	#4
Each Disk only $29.95	PMOD/CMD (disk/file memory mod.)	PCHECK/CMD PFIX/CMD	PFIND/CMD PCOMPARE/CMD
#5	#6	*LOOK*	Special!
PREFORM/CMD PVU/CMD PERASE/CMD	PCLEAR/CMD PSS/CMD PMAP/CMD	Buy TWO LDOS Utility Disks GET ONE FREE!	ALL SEVEN DISKS for $149.95
#7	#8		
PMOVE/CMD PDIRT/CMD PASSGO/CMD PUN/CMD PUN/CMD	PHELP/CMD PMX/CMD (#8) PBBOT/FIX CODE/JCL DECODE/JCL	PFILT/FLT DVORAK/FLT DVORAK/JCL	Super Utility +PLUS+ preconfigured for LDOS! $74.95 order SU+/L

All by: Kim Watt

It is a BASIC program which demonstrates the basic principles needed to communicate through the RS-232-C interface. It can be used to turn the TRS80 MODEL I or III into a dumb terminal. You can use it to communicate with another computer at a rate of 300 baud. Wonder how it works ? Here's how ! Let's expand the above 4 BASIC statements into separate lines and add some comments. Doing so, here is what the new COM/BAS looks like:

```
01  CLS                          'CLEAR THE SCREEN
02  PRINT @ 200,"COM/BAS WRITTEN BY EARL TERWILLIGER"
05  CMD"T                        'DON'T INTERRUPT ME
10  OUT 232,0                    'MODEM STATUS REGISTER RESET
20  OUT 233,85                   'SET BAUD RATE TO 0
30  OUT 234,165                  'SET RS-232-C STATUS REGISTER
31  REM 7-BIT WORD, EVEN PARITY, 1 STOP BIT
32  REM TURN ON DATA TERMINAL READY
40  X = INP(234)                 'HAS A CHARACTER ARRIVED
50  IF X < 128 THEN 100          'FROM THE RS-232-C?
55  REM                          'YES = CONTINUE  NO = CHECK KEYBOARD
60  X = INP(235)                 'GET CHARACTER FROM THE RS-232-C
65  IF X = 10 THEN 40            'IGNORE LINE FEED CHARACTERS
70  PRINT CHR$(x);               'DISPLAY CHARACTER ON THE SCREEN
80  GOTO 40                      'CHECK FOR MORE INCOMING CHARACTERS
100 X$ = INKEY$                  'DID SOMEONE TYPE IN A CHARACTER ?
110 IF X$ = "" THEN 40           'IF NOT GO CHECK THE RS-232-C
120 OUT 235,ASC(X$)              'IF SO, SEND THE CHARACTER TO THE
121 REM RS-232-C TO SEND OUT
130 GOTO 40                      'BACK TO CHECK FOR INCOMING DATA
140 END                          'END THE PROGRAM
```

Since BASIC is sometimes a little too slow to keep up with the baud rate of 300, you may notice that I added statement 5. If you do not add this statement, you may notice a missing input character or two. As you can tell, the logic of the program alternates between checking the RS-232-C and the keyboard for any input characters. If a full character is received from the RS-232-C, (50 IF X < 128 THEN 100) it is displayed on the video screen. If a character is typed in it is sent as output to the RS-232-C.

If you prefer assembler language (machine code) for its speed, take a look at COMM1/ASM. It is a Z80 assembler program for the TRS80 MODEL I or III. Its logic is essentially the same as COM/BAS. COMM1/ASM, however, recognizes the BREAK key to end and also displays the keyboard characters as they are typed in (half duplex mode). If you would like to try this program and don't have an assembler, try COMM1/BAS. It is a BASIC program which POKEs the assembled version of COMM1/ASM into memory. Try them on your TRS80 MODEL I or III !!

```
00010 ;COMM1/ASM
00020 ; WRITTEN BY EARL TERWILLIGER
00030 ;
00040 ORG     7D00H
00050 COMM1:  DI                          ;DISABLE INTERRUPTS
00060         LD
00070         OUT     (0E8H),A            ;RESET MODEM STATUS REGISTER
00080         LD      A,85
00090         OUT     (0E9H),A            ;BAUD RATE GENERATE
00100         LD      A,165
00110         OUT     (0EAH),A            ;SET RS-232-C SWITCH SETTINGS
00120 RSIN:   IN      A,(0EAH)            ;CHECK FOR RS-232-C INPUT
00130         BIT     7,A                 ;ANY INPUT
00140         JP      Z,KEYIN             ;NO. TEST KEYBOARD
00150         IN      A,(0EBH)            ;YES GET CHARACTER
00160         CP      0AH                 ;LINE FEED ?
```

```
00170         JP      Z,RSIN          ;YES IGNORE
00180         CALL    033H            ;NO. DISPLAY
00190         JP      RSIN            ;RETURN FOR MORE
00200 KEYIN:  CALL    02BH            ;ANY INPUT FROM KEYBOARD ?
00210         OR      A               ;TEST IT
00220         JP      Z,RSIN          ;NO. GO TEST RS-232-C
00230         CP      01H             ;BREAK ?
00240         JP      Z,END           ;YES END.
00250         OUT     (0EBH),A        ;NO. SEND IT TO THE RS-232-C
00260         CALL    033H            ;DISPLAY THE DEPRESSED KEY
00270         JP      RSIN            ;GO CHECK RS-232-C FOR INPUT
00280 END:    EI                      ;ENABLE INTERRUPTS
00290         RET
00300         END     COMM1
```

```
10  CLS
20  PRINT @ 200,"COMM1/BAS WRITTEN BY EARL TERWILLIGER"
30  DEF USR1 = &H7D00
40  FOR X = 32000 TO 32054
50  READ A: POKE X,A: NEXT X
60  CLS
65  PRINT CHR$(14)
70  REM                     ADDR OPCODE LABEL INSTRUCTION
80  A = USR1(0):          '                   ORG 7D00H
90  DATA 243:             ' 7D00 F3   COMM1: DI
100 DATA  62,0:           ' 7D01 3E00         LD A,0
110 DATA  211,232:        ' 7D03 D3E8         OUT (0E8H),A
120 DATA  62,85:          ' 7D05 3E55         LD A,85
130 DATA  211,233:        ' 7D07 D3E9         OUT (0E9H),A
140 DATA  62,165:         ' 7D09 3EA5         LD A,165
150 DATA  211,234:        ' 7D0B D3EA         OUT (0EAH),A
160 DATA  219,234:        ' 7D0D DBEA RSIN:   IN A,(0EAH)
170 DATA  203,127:        ' 7D0F CB7F         BIT 7,A
180 DATA  202,33,125:     ' 7D11 CA217D       JP Z,KEYIN
190 DATA  219,235:        ' 7D14 DBEB         IN A,(0EBH)
200 DATA  254,10:         ' 7D16 FE0A         CP 0AH
210 DATA  202,13,125:     ' 7D18 CA0D7D       JP Z,RSIN
220 DATA  205,51,0:       ' 7D1B CD3300       CALL 033H
230 DATA  195,13,125:     ' 7D1E C30D7D       JP RSIN
240 DATA  205,43,0:       ' 7D21 CD2B00 KEYIN: CALL 02BH
250 DATA  183:            ' 7D24 B7           OR A
260 DATA  202,13,125:     ' 7D25 CA0D7D       JP Z,RSIN
270 DATA  254,1:          ' 7D28 FE01         CP 01H
280 DATA  202,53,125:     ' 7D2A CA357D       JP Z,END
290 DATA  211,235:        ' 7D2D D3EB         OUT (0EBH),A
300 DATA  205,51,0:       ' 7D2F CD3300       CALL 033H
310 DATA  195,13,125:     ' 7032 C30D7D       JP RSIN
320 DATA  251:            ' 7D35 FB     END:  EI
330 DATA  201:            ' 7D36 C9           RET
340 END
```

In past issues of the LDOS QUARTERLY you saw some very good articles in RTC (Roy's Technical Corner). I finally got around to reading most of them. WOW ! SECRETS OF THE UNIVERSE! I decided to put to use some of the interesting things I learned from reading Roy's articles. I came up with the following program. It, as the comments tell you, demonstrates relocatibility, TCB'S and the LDOS command parse routine. (The equate vectors as shown are for MODEL I LDOS. They need to be changed for the MODEL III.) Assemble the program below and call it ALIVE/CMD. (I like and use EDAS. Great stuff Roy!) To execute it type:

```
ALIVE (ON)
```

You will see ALIVE appear and disappear character by character on the top right of the video screen. The LDOS SYSTEM(ALIVE) command does essentially what this program does. (NOW you have some source code!) To turn it off, type:

ALIVE (OFF)

The SYSTEM(ALIVE) command uses TCB slot 3, I use the unassigned slot number 0. I hope you find this program of value as I have of ROY's Technical Corner.

```
00100          TITLE  <SYSTEM ALIVE INDICATOR>
00110 ;
00120 ; ALIVE/ASM   SYSTEM ALIVE INDICATOR
00130 ;             WRITTEN BY EARL C. TERWILLIGER JR.
00140 ;
00150 ;             DEMONSTRATES  :
00160 ;                                PROGRAM RE-LOCATION
00170 ;                                ADDITION OF A TCB
00180 ;                                COMMAND BUFFER PARSE
00190 ;
00200          ORG    9000H          ;PROGRAM ORIGIN
00210 @ADTSK  EQU    4410H          ;TCB ADD ROUTINE
00220 @RPTSK  EQU    4416H          ;TCB REPLACE ROUTINE
00230 @RMTSK  EQU    4413H          ;TCB DELETE ROUTINE
00240 HIGH$   EQU    4049H          ;HIGH MEMORY POINTER
00250 @PARAM  EQU    4476H          ;COMMAND BUFFER PARSE
00260 DOS     EQU    402DH          ;LDOS RETURN ADDRESS
00270 DISP    EQU    033H           ;DISPLAY BYTE ON VIDEO
00280 START   EQU    $              ;START ADDRESS OF ALIVE
00290          LD     DE,TABLE       ;PARAMETER TABLE ADDRESS
00300          CALL   @PARAM         ;PARSE COMMAND BUFFER
00310          JP     Z,PARMOK       ;PARSE WAS SUCCESSFUL;
00320          JP     DOS            ;PARSE HAD TROUBLES
00330 PARMOK  EQU    $
00340          LD     HL,0000H       ;ADDR REPLACED BY PARSE
00350          LD     A,L            ;TEST FOR ON
00360          OR     H
00370          JR     NZ,TASKON      ;PARSE SAID ON
00380 PARMOF  EQU    $
00390          LD     HL,0000H       ;ADDR REPLACED BY PARSE
00400          LD     A,L            ;TEST FOR OFF
00410          OR     H
00420          JR     Z,PERROR       ;PARM ERROR
00430          LD     A,0            ;TCB SLOT ==> 0
00440          CALL   @RMTSK         ;REMOVE TCB
00450          JP     DOS            ;RETURN TO LAND OF OS
00460 PERROR  EQU    $
00470          LD     HL,PMSG        ;PARAMETER ERR MSG
00480 LOOP    EQU    $
00490          LD     A,(HL)         ;GET MSG BYTE
00500          OR     A              ;TEST FOR END
00510          JP     Z,DOS          ;BACK TO DOS IF END
00520          CALL   DISP           ;DISPLAY
00530          INC    HL             ;NEXT MSG BYTE
00540          JR     LOOP           ;TEST FOR END
00550 TASKON  EQU    $
00560          LD     HL,(HIGH$)     ;HIGH
00570          LD     BC,LAST-FIRST  ;LENGTH OF TCB
00580          XOR    A              ;CLEAR A
00590          SBC    HL,BC          ;HL ==> LENGTH OF TCB
00600          LD     (HIGH$),HL     ;PROTECT NEW TCB
00610          INC    HL             ;POINT TO FIRST FREE BYTE
00620          PUSH   HL             ;SAVE ITS ADDRESS
```

```
00630          INC    HL             ;
00640          INC    HL             ;
00650          LD     (TCB),HL       ;SAVE ADDRESS+2 FOR TCB
00660          POP    HL             ;RESTORE ORIGINAL ADDRESS
00670          DI
00680          EX     DE,HL          ;DESTINATION FOR TCB MOVE
00690          LD     HL,FIRST       ;START OF TCB ROUTINE
00700          LDIR                  ;MOVE IT!
00710          EI
00720 BEGIN    LD     DE,(TCB)       ;TASK ADDR IN HIGH MEMORY
00730          DEC    DE
00740          DEC    DE             ;TCB ADOR IN HIGH MEMORY
00750          LD     A,0            ;TCB SLOT NUMBER
00760          CALL   @ADTSK         ;ADD TCB TO CHAIN
00770          JP     DOS            ;RETURN TO MAGIC LAND
00780 FIRST    EQU    $
00790 TCB      DW     0000H
00800 TASK     CALL   @RPTSK         ;REPLACE TCB
00810          LD     A, 'A'
00820          LD     (3C3BH),A
00830          CALL   @RPTSK         ;REPLACE TCB
00840          LD     A,'L'
00850          LD     (3C3CH),A
00860          CALL   @RPTSK         ;REPLACE TCB
00870          LD     A,'I'
00880          LD     (3C3DH),A
00890          CALL   @RPTSK         ;REPLACE TCB
00900          LD     A,'V'
00910          LD     (3C3EH),A
00920          CALL   @RPTSK         ;REPLACE TCB
00930          LD     A,'E'
00940          LO     (3C3FH),A
00950          CALL   @RPTSK         ;REPLACE TCB
01060          LD     A,' '
00970          LD     (3C3BH),A      ;ERASE ALIVE
00980          LD     (3C3CH),A
00990          LD     (3C3DH),A
01000          LD     (3C3EH),A
01010          LD     (3C3FH),A
01020          JR     TASK           ;START OVER
01030 LAST     EQU    $
01040 TABLE    EQU    $
01050          DEFM   'ON    '
01060          DEFW   PARMOK+1
01070          DEFM   'OFF   '
01080          DEFW   PARMOF+1
01090          NOP
01100 PMSG     DEFM   'ALIVE PARAMETER ERROR !'
01110          DEFB   00H
01120          END    START
```

This is the BINHEX listing for the ALIVE program. The checksum is *50

```
11 9A 90 CD 76 44 CA 0C 90 C3 2D 40 21 00 00 7D B4 20 1D 21 00 00 7D B4 28 08 3E 00 CD
13 44 C3 2D 40 21 AB 90 7E B7 CA 2D 40 CD 33 00 23 18 F5 2A 49 40 01 40 00 AF ED 42 22
49 40 23 E5 23 23 22 5A 90 E1 F3 EB 21 5A 90 ED B0 FB ED 5B 5A 90 1B 1B 3E 00 CD 10 44
C3 2D 40 00 00 CD 16 44 3E 41 32 3B 3C CD 16 44 3E 4C 32 3C 3C CD 16 44 3E 49 32 3D 3C
CD 16 44 3E 56 32 3E 3C CD 16 44 3E 45 32 3F 3C CD 16 44 3E 20 32 3B 3C 32 3C 3C 32 3D
3C 32 3E 3C 32 3F 3C 18 C2 4F 4E 20 20 20 20 0D 90 4F 46 46 20 20 20 14 90 00 41 4C 49
56 45 20 50 41 52 41 4D 45 54 45 52 20 45 52 52 4F 52 20 21 00
```

MIXING NEWSCRIPT, ELECTRIC WEBSTER, LDOS, AND SOLE

J.L. Latham/l409 Evergreen Cir/Midwest City, OK 73110

I originally wrote documentation similar to this for ProSoft, and it is at their suggestion that I am passing it on to you through the LDOS Quarterly. If you aren't familiar with the name ProSoft, they are the distributors of one of the finest word processors available for the Model I and III computers; NewScript. Without any patching this word processor will run under all major DOS' (of course you are only concerned with LDOS), and supports most popular printers, again, without patching. It contains features found in no other WP package as of this writing, including the ability to be used by persons handicapped to the point of only having one hand, or even only one digit (finger or thumb) to type with!

This article is aimed primarily at Model I owners who desire to move their NewScript 7.x files to a LDOS double density system disk. To further limit discussion I am going to write this with the thought that those users have the following equipment and software available to them: LDOS 5.1.x, SOLE by Roy Soltoff, at least 2 5-1/4" drives that are capable of accessing at least 40 tracks, and some sort of double density board in their interface. Of course it is assumed that they have a copy of NewScript, and optionally Electric Webster (a fantastic word spelling checker). Users without SOLE will be told how to use the procedures via a single density "setup" disk, but to paraphrase Roy in his SOLE documentation "why ain't you got SOLE?". Ok, on with it.

If you follow these instructions, you will end up with a disk containing NewScript that will, with SOLE, boot on a double density system disk right into the MENU portion of NewScript. If you have double density without SOLE, then put the quarterly down, go get a copy of it, and come back and finish reading this. Seriously, if you don't have the SOLE program but do have double density, the only difference in the end product will be that you will have to boot a single density LDOS disk that has NS/CMD, NSINIT, and a STARTUP/MIN file for LDOS from NewScript on it. SOLE is so much easier.

As I stated earlier I am assuming that you have at least two (2) 5-1/4" drives or their equivalent available. These operations are more easily accomplished on a 3 drive system, but I realize that not all owners have that many. Actually, with the state of the art as it is now in word processing, it is almost a requirement to have 3 drives or more to make use of things like a full blown word processor along with one of the spelling checkers and, in the case of NewScript, a compatible Graphics program such as G.E.A.P., along with any other utilities you want to have on line.

Enough gabbing, let's do it!

IMPORTANT NOTE** Quite often throughout this writing I tell you to copy files from a disk such as "your NewScript" or "your Electric Webster" or even "your LDOS" disk. I do not mean for you to use your ORIGINAL master disk. This should be a COPY of your original. This is for your own protection. You should always work from a backup! If the software distributor doesn't think the program is valuable enough to be backed up (for YOUR protection and convenience) then perhaps you ought to wonder if it was even worth purchasing in the first place. DEMAND backup ability. You purchase the right to use a program, and if your disk goes crump in the night (or day) and you have to wait for the distributor to replace it, then you have been denied the right that you purchased for the period that it takes to get another copy!

STEP 1: Make a 40 track SINGLE density backup of your LDOS 5.1.x disk.

STEP 2: Put your original copy of LDOS back in its safe place, put the new copy in drive 0. If you have a switchable lowercase modification put the switch in the ALL CAPS position. Now press the reset button to boot the (new) system disk. Reply to the date prompt as usual.

STEP 3: KILL off all visible files except PDUBL or RDUBL and any RS-232 driver you need. The master password is in your LDOS operators manual under general information.

STEP 4: Copy the SOLE1/CMD and SOLE2/CMD files from your SOLE disk to the LDOS system disk. Put away the SOLE disk.

STEP 5: Copy the following files from the NewScript disk(s) to your LDOS disk: NS/CMD, PROP/CIM, FEDIT/CIM, NSINIT, and, from side 2, NSINSAL. MAKE SURE THAT YOUR NEWSCRIPT DISK(S) HAVE A WRITE PROTECT TAB ON THEM!.'!

STEP 6: Go to BASIC with the following command:

```
LBASIC (F=4,E=N) RUN"NSINSTAL"<ENTER>
```

or if you are running LDOS 5.1.1 or earlier (WHY?): LBASIC (F=4) RUN"NSINSTAL"<ENTER>

STEP 7: Read the NewScript "Release 7.0 Features Supplement", page 3, steps 6 through 10 to explain NSINSTAL. Also read each "page" of video as it is presented to you. Reply to the installation questions as required for your system. When you are done, return to LDOS with the CMD"S" command.

I need to say a few words about NS/CMD here before proceeding. NS/CMD is a keyboard driver that has been provided by ProSoft for use with NewScript. NS/CMD is not relocatable, and so we must be sure it is loaded in and protected before loading any of LDOS' special functions. It is a good driver and provides many special functions including a type ahead buffer. However, when used with LDOS it occasionally loses one or two characters during line wrap around if being used by a fast typist. It does not do this with other operating systems, and I don't know why it does with LDOS. For this reason you may elect to use the KI/DVR provided by LDOS instead of NS/CMD. There are complications with the <CLEAR> key involved in this, but it will work. If you elect to use KI/DVR then DO NOT set the MEMORY to X'nnnn' as directed below, and be sure to SET and FILTER your keyboard (*KI) as required for your use instead of typing NS - as instructed later. If you are going to use KI/DVR then you should move your lowercase switch to the lowercase position and boot your system again at this time. I will continue to present these installation instructions assuming you are going to use NS/CMD as your driver for use with NewScript.

STEP 8: We won't be using NSINSTAL again during these procedures, and we can better use the space for something else. At this point you may give the following command:

```
KILL NSINSTAL:0<ENTER>
```

STEP 9: Next we need to check on how our memory and system are set up. Type the following command:

```
MEMORY<ENTER>
```

The response should be: HIGH=X'FFFF'. If it is not then type the following command:

```
SYSTEM (SYSGEN=NO)<ENTER>
```

After everything settles down then re-boot the disk. We want no drivers, filters, or anything else in high memory at this point (no, not even R/PDUBL yet). Once you have high memory cleared out give the following LDOS commands:

```
NS -<ENTER>
MEMORY <ENTER>
```

The response will probably be in the range of X'EBEF' to as low as X'E700'. This is the point where NS/CMD starts living. We must protect all memory above this location before we do any thing else. Write down the value you just got from the MEMORY<ENTER> command. In the next command the 'nnnn' means to use the value you just wrote down to protect NS/CMD from further interference. First we must clear all memory again. Hit the RESET key again and check that the MEMORY is now set to X'FFFF'. Then give the following commands from LDOS READY:

```
MEMORY (HIGH=X'nnnn')<ENTER>   *not if using KI/DVR
followed by
PDUBL<ENTER>                   *or RDUBL<ENTER>
and finally
MEMORY <ENTER>
```

The response will probably be in the range or X'EBEF' to as low as X'E700'. Earlier (or later) versions may give slightly different responses, but should be in this general neighborhood. Mostly it depends on what version of PDUBL or RDUBL you are using. Using NewScript 7.0 and PDUBL from LDOS 5.1.3 I got a value of HIGH=X'EA2C'.

STEP 10: If you previously switched your lowercase to the uppercase only position it is now time to place it in the lowercase position. The next command we will type will (probably) be all garbage, so type VERY CAREFULLY until you can read your writing again. DO NOT do this step if you are going to use KI/DVR. If you are going to use NS/CMD then type:

```
NS -<ENTER>
```

followed by (and this should be readable):

```
SYSTEM (SYSGEN)<ENTER>
```

Note that there was a mandatory single space between the S in NS and the dash (-) that followed.

From now on anytime this (yes, still single density) disk is booted you will automatically have memory set to protect NS/CMD, PDUBL or RDUBL will be loaded, memory again reset to protect that program, and NS/CMD will be activated as your keyboard driver.

STEP 11: AT LAST, DOUBLE DENSITY: Place a blank disk in drive 1 and FORMAT it for double density and (at least) 40 tracks. All of the following file manipulations assume a 40 track disk, and that you have NS/CMD on the disk and are using it for your keyboard driver. Use whatever parameters you want for the new disk's name, password, etc. I recommend using LDOS-51x (where x is your version number), and the standard password since we will be making another BACKUP in just a few minutes.

Refer to your SOLE program manual and perform steps 3 through 7 found on pages 3, 4, and 5 of that manual. When you do step 5 in the SOLE manual you can consider the system to have already been set up earlier (during our step 10). So, when you get there you may just type "SYSTEM (SYSGEN)<ENTER>" and all will be well. If you have followed your SOLE instructions then the double density system disk should now be in drive 0. Try booting the system to make sure that it will boot in double density and configure properly. To test if NS/CMD got installed, press the <SHIFT> and <CLEAR> keys together. A pair of question marks should appear in the lower right hand corner of the screen. Press <SHIFT> and <CLEAR> again to make them go away.

You may now KILL the SOLE1/CMD file from the system disk. You don't need it anymore. You might leave SOLE2/CMD on there for future use of your own.

STEP 12: Now we will create an AUTO command that will automatically bring up NewScript when we boot this disk. Type the following line from the LDOS Ready prompt:

```
AUTO NS<ENTER>
```

Of course <ENTER> refers to that key. Notice that this time there is no space nor dash following the filespec NS. Invoked in this manner NS/CMD will initialize NewScript by "DOing" a file called STARTUP/MIN that we will create in a few minutes. That file will take you to the main menu of NewScript effortlessly. How nice.

If you inadvertently reset your system now it will probably get a file not found error. Don't worry we will clear all that up when we create the STARTUP/MIN file.

STEP 13: Place side 1 of the NewScript disk into drive #1 and copy the following files to drive 0: EDIT, SCRIPT, and (optionally) HELP. Then copy the LDOS/MIN file from side 2 of the NewScript disk. I am going to assume you copied the HELP file.

STEP 14: If you only have NewScript (and not Electric Webster) then copy the following files from a copy of side two of the NewScript disk to drive 0: FITLINE, GENINDEX, INDEX, and LABELS (if you have the labels option).

STEP 15: Owners of LDOS 5.1.1 and earlier may now skip down to step 15a. LDOS 5.1.2 users have another step to do now. LDOS 5.1.3 users who have not changed the default E=N of LBASIC may also go to step 15a, but if you have changed it to E=Y then you MUST do the following steps. Type the following LDOS command:

```
LBASIC (F=4,E=N) RUN"NSINIT"<ENTER>
```

Choose option 1 (EDIT) from the menu. Reply LDOS/MIN to the request for a filename. You will be presented a one line file. Change that line to read:

```
LBASIC (F=4,E=N) RUN"NSINIT"
```

That format is mandatory for 5.1.2 users, 5.1.3 users may just leave the line as it was if they kept the default E=N parameter for LBASIC.

Use the END option of NewScript's EDIT function to save the file back to disk, and then choose option 6 (Return to main menu), and when back at the main menu, choose option #8 to get back into LDOS.

STEP 15a: Type the following command from LDOS:

```
RENAME LDOS/MIN:0 STARTUP/MIN<ENTER>
```

If you are not going to install Electric Webster then you are done! Your LDOS double density disk version of NewScript 7.x is now ready for use. You can go merrily on your way. If you ARE going to install Electric Webster, well ... read on.

INSTALLING ELECTRIC WEBSTER: It is assumed that you will be working from readable copies of the EW disk. I will tell you what files to move when, finding them is up to you as I don't know how you made your copies.

STEP 16: Copy M/NEW and CORRECT2/NEW to drive 0 with the following commands:

```
COPY M/NEW:1 M/EW:0<ENTER>
COPY CORRECT2/NEW:1 CORRECT2/EW:0<ENTER>
```

Take note of the fact that the extent changes from NEW to EW when moving the files. This is important for proper operation.

Next copy the following files to drive 0 with the following series of LDOS commands:

```
COPY EW/CMD:1 MICPROOF/CMD:0<ENTER>
COPY CORRECT1/EW:1 :0<ENTER>
COPY ADDTODIC/EW:1 :0<ENTER>
COPY PRINTDIC/EW:1 :0<ENTER>
```

I had you rename EW/CMD to MICPROOF/CMD because this will allow it to work with NewScript 6.2 and 7.x properly. It is possible for early releases of 7.x to "choke" on the EW/CMD filename when files are being chained in Electric Webster. This takes care of this problem. If you have one of those early releases of 7.x why not update? ProSoft's updating policies are very liberal!

If you check the amount of space left on drive 0 you will find that there is obviously not enough room for the DICTx/EW files to reside on it. Those take a minimum of 119K! I say minimum because DICT3/EW is an expandable file. Now you begin to see what I meant earlier when I said a three drive system was very desirable for word processing. We will put the DICTx/EW files on the data disk(s) that you will use to hold the text that you are processing. Here is how (on a two drive system, it is much easier with three):

STEP 17: Place a blank disk in drive #1. FORMAT it for double density with, again at least, 40 tracks. Give it whatever NAME and PASSWORD you think appropriate. It will only be used as a data disk.

STEP 18: Once the disk has been FORMATted give the following commands from LDOS (this operation takes a lot of time, and several disk swaps, so for your own protection put write protect tab(s) on your EW source disk(s) and follow the prompts on the video exactly):

```
COPY DICT1/EW:1 (X)<ENTER>
COPY DICT2/EW:1 (X)<ENTER>
COPY DICT3/EW:1 (X)<ENTER>
```

You may want to FORMAT a couple or three more disks and use the BACKUP command with (X) option to make copies of this data disk for future use. You will notice that you have somewhere around 57K of space left on a disk that started with over 170K before the DICTx/EW files were placed on it. This space is sufficient for most users. I did this entire document on a system that was generated using these instructions and had plenty of room on the data disk for these files.

I hope that these instructions are useful to you and help you make better use of both LDOS and NewScript, with or without Electric Webster. Considered separately they are two very fine systems, together they provide the user with a very powerful word processing ability.

***** PARITY = ODD *****

(c) 1982 Tim Daneliuk
T&R Communications Associates

Hello again, and welcome to another exciting episode of Review Wars! I've been looking over lots of new programs for the next few issues of the Quarterly, and some of them are just outstanding! Before I get started though, a quick comment is in order. After the "soapbox" in my last column, I realized I left something out that needs to be mentioned. If I review a product in these pages, and give it substantially negative comments, the manufacturer of the product has the right to respond. So long as the response isn't obscene, threatening, or derogatory, I will include all or part of it in this column! You have the right to hear both sides of an issue, so I'll try to be as fair as possible in doing this. My address can be found below should you desire to do this. By the way, reader opinions are also solicited. If you have something to say about a product I review, feel free to drop me a line or leave a message on MNET.

I should also mention that I intend to use the last column as a basis for reviewing all software. I will be brutal if necessary! As I said then, there is no excuse for sloppy applications code when it is being commerically marketed.

The first item in this issue concerns a piece of hardware that is near and dear to us all, disk drives. As you all are probably aware, the cost of both bare drives and packaged systems has dropped dramatically in the last year or so. You have probably seen the ads in various magazines which promote complete 40 track drive packages selling for between $175 and $200. These packages seem to be well built and feature the 5" BASF disk drive assemblies.

Now for the sad news: Of the three people I know who have purchased BASF drives (either the above packaging or bare drives) every one has had trouble with them. The problems have all been of an intermittent nature, and seem to be caused by the drive electronics. Obviously, a half dozen or so drive failures is hardly grounds for condemning an entire product line. I mention this here only to point out that you should check around before you buy one of these units. It could well be that the failure rates here in Chicago are very unusual, and that the BASF products are just fine. There IS one major design flaw in these drives that I have noticed, however. The drive is very unforgiving if you don't push the media in all the way before closing the door. The door assembly and media guides are poorly designed, and closing a door when the media is not EXACTLY in place causes massive crunching and mangling of the disk. My own bias is toward Shugart SA400s, but I'm told that the Tandons are also very good. In the interest of getting a little field data, I hereby announce the first PARITY = ODD reader poll! Please send me a sheet of paper with the information below. If I get enough responses (more than 50) I'll publish the results in this column. You can also leave the info. for me on the LDOS board in MNET.

PARITY = ODD Poll #1: 5" Floppy Disk Drives

Name, Address, Phone #

Type of Computer

Model I Users: Type of Interface

Type, age, and number of disk drives:
(Please be specific. Don't give the packager's name, but the actual drive manufacturer. i.e Shugart SA400, instead of Lobo or Radio Shack. Also, please give the approximate age of each drive.)

Failure Rate for each drive:

1) Never failed
2) Failed after heavy use
3) Failed soon after purchase (less than 1 year of average use)
4) Continuing failure problems (more than once a year when subjected to average use)

Note: It is positively NOT a failure if a drive acts "flakey" with heads that you never have bothered to clean! A "failure" is defined to mean any problem which caused the loss of use of the drive because it had to be serviced.

Despite all the MTBF (Mean Time Between Failure) data published by the manufacturers, this kind of information from actual field experience is a lot more useful. I hope you'll all take the time to fill out a postcard and send it to me. My address is:

Tim Daneliuk
4927 N. Rockwell St.
Chicago, IL 6$625

MNET# 7$745,152$

The first product this issue is not really software or hardware, but "brainware". I refer, of course, to the best TRS-80 specialty magazine presently published, THE ALTERNATE SOURCE (TAS). This little unassuming bi-monthly booklet is a "gold mine" of TRS-80 related (especially Mod. I and III) information. For those of you who have not been subscribing, TAS back issues are available in bound form. These include rather thorough treatments of how the Level II BASIC Interpreter works, explanations of bugs in the ROM, assembly language technique, and a host of other subjects. If you own a TRS-80, this is the one technical magazine you should get. It doesn't have a lot of ads, but is chock full of useful "goodies". The address for TAS is:

The Alternate Source
704 North Pennsylvania Ave.
Lansing, MI 48906

(517) 482-8270 or (800) 248-0284
MNET# 70150,255 SOURCE# TCH565

TAS also publishes and distributes software, and this leads us to our first software product this Quarterly.

MODEM 80 is a program distributed by TAS as well as it's author, our own Les Mikesell (take a bow Les!). It is a general purpose serial communications program, which turns the TRS-80 into a "smart" terminal. The cost is a mere $39.95, and a special LX80 version is available for $10 additional. If you only use serial communications occasionally, LCOMM is probably all you need, but if you do a lot of it, you should investigate MODEM 80. One of it's biggest features is that it honors the file transfer protocols used by CP/M bulletin boards. Another feature I particularly like is that MODEM 80 allows you to maintain simultaneous file transmit and receive buffers. Les has also included several useful utilities with the package (some of these won't work with an LX80 so check before you buy!). There is a program similar to the Model II HOST utility which allows remote control of your TRS-80 via the serial port. XMODEM is a utility to transfer files. SAVE and TYPE are used to create and list text files. TEXTFIX is used to fix downloaded files so they load into a word processor. It strips out control characters, extra linefeeds, and deletes characters caused by a backspace. TEXTFIX can also add a X'00' to the end of the file. Finally, HEX is a utility used to convert ASCII files to hex characters and vice-versa. This is a fine program written by an excellent TRS-80 programmer! I have found it easier to use than LCOMM in many instances, and you "Fhone Phreaks" will just love it.

As mentioned in a previous column, I intended to do a comparitive review of the popular spelling checker programs for the TRS-80. Unfortunately, they have not all arrived in time for a simultaneous review. So, I will do them one at a time, and make some comparisons when they have all been reviewed. Again, the emphasis will be primarily on LDOS compatibility, though I'll try to make some judgements on the overall performance of these products. The first of these is HEXSPELL II from Hexagon Systems. HEXSPELL is written in BASIC, but is distributed as an executable file generated by the Microsoft BASCOM BASIC compiler. Once the program is loaded and the file to be checked is named, the text of that file is displayed on the screen. You may select the scrolling rate, and there is also a command which stops the display for ease of reading. Any time HEXSPELL finds a word it does not recognize, you have three choices: 1) You can ignore the word, and HEXSPELL will leave it "as-is" without appending it to the dictionary. 2) You can tell HEXSPELL to learn this word as a new word, and add it to the dictionary file. 3) You can replace the word with the correct spelling. HEXSPELL inherently shows words "in context" because a portion of the total text is always on screen. The word in question is easy to find because it is highlighted with a flashing graphics block. The program is extremely easy to use because it is very interactive. In fact, for most of the functions you don't even need the manual. The manual is about 30 pages long, and is very well written.

There are several things about HEXSPELL II I don't like. For one thing, because of the large runtime support package (BRUN) that BASCOM compiled programs require, HEXSPELL is a real "memory hog". It seems to honor HIGH$ alright, but if it decides there is not enough memory available, it forces an error message. Now, all this is fine, except that the error message asks you to press any key to continue. When you do, the program tries to load again and the same error message occurs. In other words, once you get an "Insufficient Memory" error, you can't get out of HEXSPELL and back to LDOS! I solved this problem by always using HEXSPELL with no high memory options installed. This is a real pain though! I've been using a LOBO hard disk with my LX80, and to access all the hard disk partitions the LDOS drivers have to be in high memory.

Anytime I want to use HEXSPELL, I have to boot the system, and then either use a JCL procedure or manually install the hard disk drivers. This is a problem that ought to be corrected because it degrades an otherwise fine program. Another minor complaint I have is that HEXSPELL runs rather slowly. This is a price that is paid by writing a program in BASIC (even though it is compiled), and making that program very interactive and easy to use.

On the positive side, HEXSPELL has several excellent features. It is easy to use, corrections are simple, and it runs bug-free (once high memory has been freed). Another very important asset of HEXSPELL is that files to be checked are not limited by available memory in the system. HEXSPELL loads in a portion of the file at a time for checking. This allows you to check files created by a disk oriented word processor or text editor.

At $99, HEXSPELL is priced well below competing products. Except for the one problem mentioned above, the product has run perfectly, and it is very well integrated with LDOS. A new release of the product is presently being worked on which will free about 2K more of memory, so the high memory problem should be minimized. You should evaluate your needs carefully before deciding for or against HEXSPELL. On the one hand, it's ease of use makes it an ideal candidate for an office where non-technical people need to use a spelling checker. On the other hand, it's slowness will be a real drawback for someone who writes a great deal. HEXSPELL II is available from:

Hexagon Systems
P.O. Box 397
Station A
Vancouver, B.C. V6C 2N2
Canada

(604) 682-7646
MNET# 70235,1376

Next on the list of software "goodies", is the set of utilities for LDOS from Powersoft. These utilities were all written by Kim Watt of Super Utility fame. There are seven utility disks available, and all are designed to be specifically LDOS compatible. The documentation for these utilities is printed on standard three-hole punched paper, designed to go into your LDOS binder. Each of the disks costs $29.95, though a special price is available if you order the whole set.

I can't cover all these utilities here, so I'll just mention the best of the bunch. PMOD is a general purpose Disk-File-Memory modification utility. It allows you to access the disk absolutely, or to access it on a file by file basis. You can also examine and modify memory directly with PMOD. PCHECK and PFIX are utilities which allow you to check and (if necessary) repair a damaged directory. These two utilities have saved me a LOT of time! PFIX attempts to repair both the HIT and GAT sectors, and you can even copy a new BOOT file onto a bad disk. I have had several unreadable disks which were fixed with PFIX. These programs are definitely not "idiot-proof" though, and should be used carefully. PREFORM allows you to take a 5" floppy disk and reformat it without loosing the data it contains. This procedure can sometimes help marginal disks which have not been formatted in a long time. PSS and PMAP are utilities which help you identify which sectors on a disk have been allocated, and to which files they have been assigned. PCLEAR will erase unassigned sectors on a disk so no trace of the original data is left. You can also use PCLEAR to erase unused directory entries. PHELP is an "on-line" help file. I found it to be a rather complete disappointment! When I asked for help with extended DEBUG. All I got was:

```
DEBUG (switch,EXT
switch  ON or OFF.  ON is assumed
EXT turns on the extended debugger
```

Not real helpful, eh? PHELP is on Disk #8, and frankly, I can't see much on this disk that warrants your money. There are a few specialized filters including a translation filter for the keyboard and a filter to change the LDOS graphics on boot up. The programs on this disk seem to work alright, but they just aren't terribly useful.

On the whole these programs seem to work smoothly with LDOS. I did have problems with several programs including PCHECK, PFIX, PSS, and PMAP. They worked fine on both 5" and 8" floppies, but refused to run on the LOBO 1850 hard disk. A call to Powersoft indicated that they had no problems on their Laredo system, but that they would look into this apparent incompatibility. One especially nice feature that all these utilities have, is that they have help built right into them. Once the program is loaded, all you do is hit <ENTER>, and the screen displays the possible sub-commands, sometimes with an example (if space permits). This is a GREAT idea. I hope everyone "steals" it from them, and it becomes an industry standard!

I doubt any one user needs all of these utilities, but there are several you shouldn't be without. PFIX and PCHECK are winners (hopefully the hard disk problem will be resolved), as is PMOD. Stay away from Disk #8. If you need translation filters, buy the filter disk from LSI. It has more filters which are more usable than these from Powersoft. If you really want to change your boot graphics, use FED to change them in SYS0. The remaining utilities will have varying degrees of usefulness depending on your application. You should study your needs carefully before ordering. These are, by and large, good products, and you won't be disappointed. These utilities are available from:

POWERSOFT
11500 Stemmons Expressway, Suite 125
Dallas, TX 75229

(214) 484-2976
MNET# 70130,203

The final product in this column is the new HELP/QRC product from Misosys. This is an "on-line" Help utility for LDOS accompanied by a Quick Reference Card (QRC). There are several help files included for the "A" Library, "B" Library, LDOS Syntax, Utilities, and so on. Each of these uses the Misosys Partitioned Data Set (P0S) format, and programs are included to create your own Help PDS. One word of warning, HELP/QRC requires the Lower Case modification in a Model I. This is a fine product, which I find myself using all the time. The information for each command is not exhaustive, but IS useful. The QRC is also quite helpful, and at $25 this package is reasonably priced. HELP/QRC is available from:

Misosys
P.O. Box 4848
Alexandria, VA 22303-0848

(703) 960-2998

Well that's it for this time. Don't forget to send in your responses to the disk drive poll! Happy bit-twiddling...

....................er....................

by Earle Robinson

This issue will concentrate on a few of the more commonly encountered time and space wasters that I have seen in many assembly programs. One of the most frequently seen arises from many authors' ignorance of the use of terminating bytes at the end of a text to be displayed (or printed). A friend sent me the source for a program which looked like this:

```
          LD      HL,MSG1
          CALL    @DSPLY
          LD      HL,MSG2
          CALL    @DSPLY
          LD      HL,MSG3
          CALL    @DSPLY
          several more Loads, and displays
```

And, the screen had several lines of my friend's menu displayed. What he did not know was that he could merely have terminated each line, not with a carriage return (0D hex) or with 03, but with whatever text to be displayed, and that the display handler would continue, printing out each line as encountered until either 0DH or 03 was encountered. So, the display's source should have looked like this:

```
          LD      HL,MSG
          CALL    @DSPLY
```

The recently published SuperSCRIPSIT contains some code which looks like this:

```
          LD      A,(7A01H)
          LD      (790AH),A
          LD      A,(7A02H)
          LD      (790BH),A
          LD      HL,(9322H)
          program continues
```

Now, why didn't the author merely load the contents of 7A01H into HL, and then load the contents of HL into 790BH, thereby saving 6 bytes? There are several examples of this wasteful type of writing in SuperSCRIPSIT, which by the way, is full of other ghastly examples which I may draw upon in the future.

Another typical error committed by inexperienced assembly language writers is the following, illustrated in a patch to SuperSCRIPSIT which was recently published:

```
          LD      HL,(4514H)
          LD      (5B14H),HL
          more code
          NOP                 ; This is at 5B14H
          NOP
          LD      DE,(5B14H)
          more code
```

The writer of that code could have more economically loaded the saved address at 4514H as follows, saving 3 bytes of available patch space:

```
          LD      DE,0        ; two bytes where
                              ; address is to be
                              ; saved.
```

On a more positive point, I'd like to briefly discuss the macro capabilities now available in Roy Soltoff's new version of EDAS, called EDAS IV. Many of you may have used macros while working with the Microsoft M80. With the new EDAS you can save a lot of time, and extra source code by developing some simple routines or blocks of code which are frequently used. For example, it is often necessary to save the registers, and the code not only must be typed in, but must be remembered so that they are popped in the reverse order upon exiting the routine.

With the new EDAS I have created two macros as follows:

```
PUSHREG   MACRO      #PARM1=HL,#PARM2=DE,#PARM3=BC
PUSH      #PARM1     ; It is possible to replace
PUSH      #PARM2     ; the default values if
PUSH      #PARM3     ; required
ENDM

POPREG    MACRO      #PARM1=BC,#PARM2=DE,#PARM3=HL
POP       #PARM1
POP       #PARM2
POP       #PARM3
ENDM
```

You can also make up many others to save time in looking up the code required, or avoiding an error in writing it due to faulty memory. An example of the type of small routine which I prefer to invoke with a macro rather than type it in each time, concerns the checking if the user has used a valid drive number, from 0 to 7.

```
DRIVEOK   MACRO      #GETKEY
CP        38H
JR        NC,#GETKEY ; >7, return & ask again
CP        30H
JR        C,#GETKEY  ; <0, go ask again
ENDM
```

Since it is also possible to list the source without the macros being expanded, it is much easier to study source listings to determine if there may be bugs, or to find them if they do occur. Roy has done a terrific job with his new version of EDAS. Anyone who wishes to do serious assembly language work could do worse than make the investment.

It is not only important to well document one's source code when writing assembly language programs, but it is also most important to keep track of versions of what is written. Recently, someone who acquired my disk catalog program, 'discater', said he had problems in knowing which version of a particular program was on disk, and could 'discater' differentiate between them. Of course, since each entry is shown with its number of bytes, if one is longer or shorter than another, it would show up on a full listing. However, as I pointed out to my correspondent, you should ALWAYS modify names of programs for different versions. This will save much time in debugging and in locating the version which is the one which you really wish to use or work on. You will also avoid problems by automatically updating the date in the source before saving it to disk, by the way.

ITEMS OF GENERAL INTEREST

Following are the TCHRON and T-TIMER patches for use with LDOS 5.1.3. Apply these patches to SYS0/SYS.

```
. TCHRON1, TRSWATCH, TIMEDATE80 - /FIX - 08/19/82 - Copyright by Roy Soltoff
. This FIX is to enable use of the above clock modules
. under LDOS Version 5.1.3 Model I. Use the HI ports.
.
.  Fix the timer interrupt routine
D04,08=D2 45 ED 78 0D CD A6 47 ED 78 0D E6 0F 85 12 1B
D04,18=C9 11 43 40 01 B5 03 CD C3 45 10 FB
. Fix the date initialization on BOOT
D0D,58=21 46 40 01 BA 01 CD C3 4E 06 03 CD C3 4E 01 BC
D0D,68=0F CD C3 4E EB DB BB E6 03 11 45 40 21 48 40 20
D0D,78=29 CB FE 18 25 ED 78 0D A0 07 57 07 07 82 57 ED
D0D,88=78 0D E6 0F 82 77 2B C9
. end of patch
```

```
. T-TIMER/FIX - 08/19/82 - Copyright by Roy Soltoff
. This FIX is to enable use of the TTIMER clock module
. under LDOS Version 5.1.3 Model I. Use the HI ports.
.
. Fix the timer interrupt routine
D04,08=D2 45 ED 78 0D CD A6 47 ED 78 0D E6 0F 85 12 1B
D04,18=C9 11 43 40 01 C5 03 CD C3 45 10 FB
. Fix the date initialization on BOOT
D0D,58=21 46 40 01 CA 01 CD C3 4E 06 03 CD C3 4E 01 CC
D0D,68=0F CD C3 4E EB DB CB E6 03 11 45 40 21 48 40 20
D0D,78=29 CB FE 18 25 ED 78 0D A0 07 57 07 07 82 57 ED
D0D,88=78 0D E6 0F 82 77 2B C9
. end of patch
```

For those of you trying to interface different clock boards, following is the assembler code that generated the above patch. The TTIMER port differences are noted in the comment field surrounded by astersiks.

```
00100 ;Tchron/ASM - 08/17/82 - Copyright 1981 by Roy Soltoff
00110 ;*=*=*
00120 ; TCHRON mods to LDOS Version 5.1.3 (Model I)
00130 ;*=*=*
00140 TIME$   EQU     4041H           ;Time string
00150 DATE$   EQU     4044H           ;Date string
00160 DCTFLD@ EQU     47A5H           ;Mask & multiply by 10
00170 LOW$    EQU     4047H           ;Extended date
00180 ;
00190 ;*=*=*
00200 ; Routines for time reading
00210 ;*=*=*
00220         ORG     45C1H           ;Normal timer routine
00230 ;
00240         DW      TIMER           ;Set timer TCB
00250 GETARG  IN      A,(C)           ;Get high byte
00260         DEC     C               ;Bump port
00270         CALL    DCTFLD@+1       ;Strip, *10 -> L
00280         IN      A,(C)           ;Get low byte
00290         DEC     C               ;Bump port
```

```
00300          AND     0FH              ;Strip
00310          ADD     A,L              ;Add in high * 10
00320          LD      (DE),A           ;Stuff value
00330          DEC     DE               ;Bump pointer
00340          RET
00350 TIMER    LD      DE,TIME$+2       ;Point to end of string
00360          LD      BC,3<8!0B5H      ;Set loop cnt & port ***0C5H***
00370 LOOP     CALL    GETARG
00380          DJNZ    LOOP
00390 ;*=*=*
00400 ; Now fall through to RET instruction at TIMHK$
00410 ; User must NOT use SYSTEM (UPDATE)...
00420 ;*=*=*
00430 ;*=*=*
00440 ; Interface to system initialization for DATE
00450 ;*=*=*
00460          ORG     4E9EH            ;Address of DATE query
00470 PACKIT   EQU     4EE8H            ;Addr of packing routine
00480 ;
00490          LD      HL,DATE$+2       ;Point to end of parm
00500          LD      BC,1<8!0BAH      ;Set mask & port ***0CAH***
00510          CALL    GETVAL           ;Get month
00520          LD      B,3              ;Set mask for day
00530          CALL    GETVAL           ;Get day value
00540          LD      BC,0FH<8!0BCH    ;Set year mask & port ***0CCH***
00550          CALL    GETVAL           ;Get year
00560          EX      DE,HL            ;Point DE to date$-1
00570          IN      A,(0BBH)         ;Ck on leap year ***0CBH***
00580          AND     3
00590          LD      DE,DATE$+1       ;Set for pack routine
00600          LD      HL,LOW$+1
00610          JR      NZ,PACKIT
00620          SET     7,(HL)           ;Set leap year bit
00630          JR      PACKIT           ;Go pack it in
00640 ;*=*=*
00650 ; Routine to input date values
00660 ;*=*=*
00670 GETVAL   IN      A,(C)            ;Get high
00680          DEC     C                ;Reduce port
00690          AND     B                ;Mask high
00700          RLCA                     ;High * 10
00710          LD      D,A
00720          RLCA
00730          RLCA
00740          ADD     A,D
00750          LD      D,A              ;Save high * 10
00760          IN      A,(C)            ;Get low
00770          DEC     C
00780          AND     0FH              ;Mask the bad stuff
00790          ADD     A,D              ;Add in high
00800          LD      (HL),A           ;Stuff in parm
00810          DEC     HL
00820          RET
00830          END     4E00H
```

Some people have requested a CLS command that would work from the LDOS Ready level so it could be used in a JCL file. To make one, use the BUILD command to create the CLS/CMD file as follows:

```
BUILD CLS/CMD:0 (HEX)<ENTER>

0202C901<ENTER>
<BREAK>
```

For Model I, 5.0.3 users, the following routine will speed up printing when using the SPOOL command. It does so by inserting another despooling call in the @KITSK vector. For proper operation, this routine MUST be executed after the SPOOL command has been given. Following is a source listing plus the BINHEX code. DO NOT use this on Version 5.1!! To turn off this function, use the short program listed after the FASTSPL code.

```
00100 ; Use on Model I, Version 5.0.3 Only!
00110 ; FASTSPL - Fast despooling 5.0 Utility
00120 ;
00130 TCB$     EQU     4500H
00140 HIGHS    EQU     4049H
00150 @EXIT    EQU     4020H
00160 @KITSK   EQU     4300H
00170 ;
00180          ORG     5200H
00190 START    LD      HL,(HIGH$)       ;Get current HIGHS
00200          LD      BC,LENGTH        ;Length of new code
00210          OR      A
00220          SBC     HL,BC            ;Sub needed space
00230          LD      (HIGH$),HL       ;Store new HIGH$
00240          INC     HL               ;Loc. new code
00250          PUSH    HL
00260          LD      HL,(TCB$+18)     ;SPOOL TCB addr.
00270          LD      (NWTCB1),HL      ;Insert TCB in new code
00280          INC     HL
00290          INC     HL               ;P/U despool addr
00300          LD      (NWTCB2),HL      ;Insert despool in new code
00310          POP     DE               ;Where new code goes
00320          PUSH    DE               ;Save for later
00330          LD      HL,NEWCOD        ;Actual new code start
00340          LDIR                     ;Move to High Mem.
00350          LD      HL,@KITSK+2      ;Put new despool into KITSK
00360          POP     DE               ; vector, so get entry point
00370          LD      A,0C3H           ;C3 = JUMP opcode
00380 ; Now install the new KITSK vector Backwards to avoid system crash
00390          LD      (HL),D           ;MSB of entry pt
00400          DEC     HL
00410          LD      (HL),E           ;LSB of entry pt
00420          DEC     HL
00430          LD      (HL),A           ;Insert JP vs. RET
00440          JP      @EXIT            ;All done!
00450 ;
00460 ; New Code to be moved into high memory
00470 ;
00480 NEWCOD   DI
00490          LD      IX,$-$           ;This will be new TCB
00500 NWTCB1   EQU     $-2              ;Stuff TCB into opcode
00510          CALL    $-$              ;Call despool
00520 NWTCB2   EQU     $-2              ;Stuff despool task addr.
00530          EI
00540          RET
00550 LENGTH   EQU     $-NEWCOD
00560          END     START
```

This is the code to turn off the FASTSPL function

```
00100 ; Turn off FASTSPL
00110 ;
00120 @KITSK  EQU     4300H
00130 @EXIT   EQU     402DH
00140 ;
00150         ORG     5200H
00160 START   LD      A,0C9H          ;RET Instruction
00170         LD      (@KITSK),A      ; to disable KITSK
00180         JP      @EXIT           ;Back to LDOS Ready
00190         END     START
```

This is the BINHEX code for the FASTSPL and the turn off program. If you are using the /CMD version of BINHEX, the checksum for the FASTSPL program is *9C. These programs may also be created by using the BUILD (HEX) command.

```
01 3A 00 52 2A 49 40 01 0A 00 B7 ED 42 22 49 40 23 E5 2A 12 45 22 31 52 23 23 22 34 52
D1 D5 21 2E 52 ED B0 2A 02 43 D1 3E C3 72 23 73 23 77 C3 2D 40 F3 DD 21 00 00 CD 00 00
FB C9 02 02 00 52
```

```
01 0A 00 52 3E C9 3A 00 43 C3 2D 40 02 02 00 52
```

When using the SPOOL Library command with a serial printer, it is important that the PR/FLT be installed. The following commands show how to establish the serial driver and the spooler.

```
SET *PR RS232x (any parameters)
FILTER *PR PR/FLT
SPOOL *PR (any parameters)
```

BASIC Concepts - The RUN,V Command

by Dick Konop

LBASIC introduces many new features to programming in disk BASIC. Among the enhancements that have been incorporated is the ability to chain programs together and allow for having variables common to more than one program. This is accomplished by using the "V" parameter of the RUN command. This article will detail the use of this parameter and will touch upon some of its possible uses.

Before we explore the specifics involved in using the "V" parameter, let us consider the general syntax used to execute an LBASIC program. The following is the syntax that is required.

```
RUN"filespec",file/variable parameter,line number
```

Filespec represents the name of the program that you wish to run, and can be represented as either a string constant or string expression.

The file/variable parameter is optional, and is used primarily to perform chaining of programs. One of two different parameters is available. If the parameter R is used, any files which are currently open will remain open when the new program is loaded and executed. If the V parameter is used, all open files will remain open, and all variable assignments will be maintained. If this parameter is used, it must be represented as a letter (R or V), and cannot appear within quotes or cannot be represented as a string expression.

The line number parameter is also optional, and is used to specify a line number in the "chained" program where execution is to start. It must be represented as a numeric constant. If both the file/variable parameter (R or V) and the line number parameter are specified, the file/variable parameter must appear physically before the line number parameter.

When using the RUN command with the V parameter, there are several points that need to be noted. In addition to all files remaining open, the fielding of the buffer associated with the open file will remain intact. Hence, refielding is not required. (NOTE: Earlier versions of LDOS 5.1 did not allow for the saving of fielded variables.)

If DEFinition statements are to be used (such as DEFINT, DEFSTR, etc.) they must be established in the first program which is run, and must be re-established in any subsequently chained programs. The CLEAR statement, if encountered in any chained program, will close any open files and destroy established variables.

If the program to be chained is longer than the calling program, or uses more variables than the calling program, an OUT OF MEMORY or OUT OF STRING SPACE error could result. It should be noted that any and all variables that have been established in the calling program will be maintained in the chained program. For this reason, forethought must be used in determining variable names that will be used when chaining programs together.

Before considering some of the uses for the V parameter, an example which illustrates its implementation is in order. Listed below are two programs that reference each other (PROG1/BAS and PROG2/BAS). The sequence can be started by issuing either of the commands RUN"PROG1/BAS" or RUN"PROG2/BAS".

```
5 'PROGi/BAS
10 CLEAR 2000
20 DEFINT A-Z:DEFSTR S
30 IF A=0 THEN S="PROG1/BAS"
40 CLS
50 A=A+5
60 PRINT"THIS IS ";S,"A=";A
70 IF A>100 THEN END
80 S="PROG2/BAS"
90 INPUT"PRESS <ENTER> TO RUN PROG2/BAS";S1
100 RUN"PROG2/BAS",V,20
```

```
5 'PROG2/BAS
10 CLEAR 2000
20 DEFINT A-Z:DEFSTR S
30 IF A=0 THEN S="PROG2/BAS"
40 CLS
50 A=A+3
60 PRINT"THIS IS ";S,"A=";A
70 IF A>100 THEN END
80 S="PROG1/BAS"
90 INPUT"PRESS <ENTER> TO RUN PROG1/BAS";S1
100 RUN"PROG1/BAS",V,20
```

Although simplistic in nature, the above example will illustrate the proper method of implementing the V parameter of the RUN command. There are two variables that are passed between the programs (A and S). Each program displays the values which are currently represented by the variables. In each program, the variable A is incremented. The chaining of the programs will continue until the variable A is assigned a value greater than or equal to 100.

There are several points that should be noted concerning the implementation of the RUN command in these two programs. The line number parameter in the RUN commands plays a key role in the chaining of these two programs. If it were not included, the CLEAR statement in each of the programs would be executed. This would cause any existing variable values to be destroyed, and would nullify the results of the chaining process. Also, execution of the chained program starts with line 20 in both instances. It is important that the DEFinition statements contained in line 20 are executed. Having DEF statements in one program and NOT in a program to be chained will lead to unpredictable results, and the values stored in chained variables will not be the same.

Now that we have discussed how to implement the V parameter of the RUN command, let us consider some practical uses of it. Chaining programs together while retaining variable values can prove to be very useful when running (or writing) an integrated applications package. A very common practice when writing interactive programs which comprise a software package is to utilize some type of index file as a means to access information in a pre-defined order. If these index files are stored in RAM, then it should be obvious that each module of the software package would have to load in the index files from disk prior to performing its operation. Similarily, each module would have to write the index files back out to disk to reflect any changes that may have occurred in the index files.

By using the V parameter of the RUN command, it is now possible to run a menu driven application in which any index files that need to be loaded can be done so at the main menu only, and then passed along to all of the supporting program modules. Likewise, these index files need only be saved back to the disk at the end of the session. Programming in this manner can save several minutes when running EACH module (not to mention that the code required to load/save the index files does not need to be duplicated in all of the modules).

Another example of why the V parameter would be desirable to use follows a similar train of thought. Many programs are written in such a manner that all functions performed are done so by one all-encompassing module. Due to machine limits in terms of the amount of available memory, this may definitely become a factor regarding the amount of items that the program can handle (particularily if index files are stored in RAM). Also, if the program is designed to take advantage of various LDOS features (such as the SPOOLER), memory constraints may become a problem. By chaining programs together, the problem of running out of memory is alleviated greatly. Since LBASIC incorporates a high speed load of programs, the time it takes to chain programs together is minimal. Writing a program that could normally fit into memory as a series of very small program modules (each designed to perform a specific task) allows for many memory related features to be incorporated, as memory constraints are virtually done away with. It is also the perfect way to increase the amount of items that your program will handle, as the size of your index array may be increased (this assumes that your media storage capabilities will allow for the additional data).

The reasons listed above for why and when to use the V parameter of the RUN command are by no means exhaustive. Many other situations and circumstances could dictate that program chaining be performed. This is also not to say that program chaining should be done in all instances. Depending on your application, program chaining is a viable alternative for writing programs that would normally be cumbersome to deal with. The basic premise behind writing interactive chained programs is that you have the capability of writing program modules which, if written as one program, would greatly exceed the limits of available memory. Also, in many instances, the time that it takes to run interactive programs can be diminished, as temporary files do not need to be created to pass information to/from various program modules. It may be well worth your while to investigate the inherent capabilities found by chaining program modules.

THE JCL CORNER

by Chuck

This month's column looks like it will be a collection of random bits and pieces about JCL. Since I've received only one suggestion for information to cover, I'm not sure whether anyone is using JCL, anyone reads this column, or everyone already knows everything they need to about JCL. Any way, to start it off . . .

As you all know (?), the JCL logical operators & (AND), + (OR), and - (NOT) can be used in //IF statements to test the logic truth of a token or series of tokens. Consider the case where you are using four tokens for some purpose, perhaps in a system initialization routine to set up the number of floppy drives in a system. The tokens Fl, F2, F3, and F4 represent a system with one to four drives, respectively. To be sure the operator has declared the number of drives, you could use a test such as:

```
//IF -f1&-f2&-f3&-f4
//. You must enter number of floppy drives!
//QUIT
//END floppy test
```

In English, the //IF reads "If not f1 and not f2 and not f3 and not f4". In other words, in the case where none of the tokens was entered, the //IF would be true and the warning message would be printed and the compiling would //QUIT. This is a perfectly valid and workable JCL logical evaluation method.

The other case to check would be if the operator entered more than one token. This next example, although it looks correct, shows how -NOT- to do it!

(the underlines are only to show the different possible combinations)

```
//IF fl&f2+fl&f3+fl&f4+f2&f3+f2&f4+f3&f4
//. Enter only one floppy number
//QUIT
//END multiple floppy test
```

By the way, the order of the two tokens in the AND group makes no difference in the LOGICAL evaluation; fl&f2 is the same as f2&fl. The logic of the //IF is valid; all multiple combinations of the four tokens are tested for. The problem lies with the method JCL uses to evaluate the & AND operator. When evaluating a conditional line, the JCL processor will stop when it finds a false token followed by an AND symbol. Thus, the //IF line in the preceding example would correctly pick up all cases where f1 and another f token were entered together, but would miss multiple combinations of the f2, f3, or f4 tokens. For example, suppose that both f1 and f3 were entered. The processing would test f1 and find it true, then test f2, making the first group false. However, it would continue and find that the next group of f1 and f3 would be true, making the //IF true. The end result would be that the //IF would be true and the job would abort as expected. If f2 and f3 were entered, the processor evaluate the first f1, find it false, and then find the & AND operator following the f1. At this point, the rest of the line would be ignored and the //IF would be false, allowing the multiple tokens to create havoc later in the JCL. The answer to this problem is the use of multiple //IF lines.

```
//IF f1&f2+f1&f3+f1&f4
*
//IF f2&f3+f2&f4
*
//IF f3&f4
*
```

* represents the three line sequence of //. error message, //QUIT, and //END.

Using this method assures that all possible combinations of multiple tokens can be caught. It makes the JCL file longer, but it also makes it work properly! You may ask why the AND evaluation is done in this manner, and why something is not done about it. Just to get some ideas for future columns, I'm not going to tell you. If you really want to know, write in and ask, including at least one idea/suggestion for future JCL Corner columns.

Continuing on, I am happy to announce that we have tracked down a bug introduced somewhere between 5.1.1 and 5.1.2 that caused the following problem. Have KI/DVR set, and have TYPE, JKL or the SPOOL function active. Do a JCL file to enter LBASIC and run a program. Have the first keyboard request in the program be an INKEY$ request. Pressing any key (except <BREAK>) has no effect, as if LBASIC was ignoring the keyboard. Press <BREAK> to break the program, then RUN the program and everything works fine. This same sequence of commands, if typed in by hand rather than executed from a JCL file, also works fine. Therefore - a BUG in JCL.

WRONG! The bug was fixed in the JCL overlay, but was caused by the changes made to the KI/DVR program! Remember the KI/DVR patches in the last Quarterly to make the //INPUT macro work properly? The need for them resulted from the fact that KI/DVR was given the ability to handle @CTL calls to flush the type ahead buffer when exiting a JCL procedure. As it turns out, the same change to KI/DVR caused the problems in this case. We re-assembled SYS11 to fix the problem, and now everything runs smoothly. As Mr. Murphy put it - make a change here and I'll get you there.

On a brighter note, Les and myself worked out a patch to LSCRIPT to allow it to be controlled (with certain restrictions) by a JCL file. The method used was to force LSCRIPT to initialize in the "SPECIAL COMMAND ?" mode. Additionally, everytime a command line is entered and an <ENTER> is detected, the command is executed and a return to the "SPECIAL COMMAND ?" mode is forced. When using this patch, the screen format may appear to be scrambled. Don't worry - the actual text is still O.K. What you are seeing is an interaction between Lscript's cursor positioning and the cursor positioning used by the @KEYIN routine to feed in a JCL line.

From this point on, I will use the name JLS/CMD to represent an LSCRIPT patched to enable the use of JCL.

The simplest use for this patch is to print a series of files whose names are contained in a JCL file. The proper sequence would be:

```
JLS/CMD
L file1
P
L file2
P
etc.
```

This set of commands is feeding in the load file (L filename) and print file (P) to LSCRIPT. It can be repeated for as many files as desired. Many applications also require that a header or footer be changed as each new file is printed. For ease of explanation, consider an example where the header text line is:

```
LDOS Hex code dump - File --> SYS0
```

For purposes of this example, I will assume that there are 13 files to be printed by this JCL, hex dumps of SYS0 through SYS12. Also, this header block is saved as a SEPARATE file named HDR/SCR for purposes of flexibility when printing. To print the files and change the header for each file, you could construct a JCL file as follows (the comments in parentheses are for explanation only):

```
JLS/CMD              (execute patched Lscript)
L HDR/SCR            (load the header, no change needed)
L,C SYS0/SCR         (Load the first file)
P                    (print it!)
L HDR/SCR            (reload header)
R>SYS0>SYS1          (change header filename)
L,C SYS1/SCR         (chain in next file)
P                    (print file)
etc.                 (do as many times as necessary)
```

The sequence of events for the first four lines should be self explanatory, but I'll detail it anyway. The patched Lscript is executed and starts looking for input. The header is loaded, and no change is needed since SYS0 is the first file to be printed. The SYS0 file is loaded and chained to the header. The resultant file is printed. For the remaining files, it will be necessary to modify the header message to reflect the proper filename. Since the R (replace) command is executed from the "SPECIAL COMMAND ?" mode of Lscript, it can be entered into the JCL file. Thus the remaining sequence of lines reloads the header (and removes the old text from memory), updates the header to the proper file name, chains in the text to be printed, and then does the print command.

Those are the basics needed to use the JCL patched Lscript. In conclusion, any line in the controlling JCL file must be a valid "SPECIAL COMMAND ?" reply. No direct modification of text is allowed.

Following is the patch for Lscript, both Model I and III. Be sure to apply this to a "virgin" Lscript - it may interfere with patches developed by yourself or other users.

```
. JCS Patch to allow JCL control of LSCRIPT
. By Les & Chuck  09/01/82
. This patch is for Models I and III
.
D00,12=3A 3C 52 B7 20 1A 06 3F 21 6A 63 CD 40 00 78 DA
D00,22=30 40 B7 28 EB 3C 3C 32 3C 52 2B 36 1D 22 3D 52
D00,32=3D 32 3C 52 2A 3D 52 7E 23 22 3D 52 B7 C9 00 00 00
D0E,66=0E 52
. EOP
```

For a final subject I am going to touch on using JCL to install keyboard filters or drivers. This may only be of interest to those of you who write your own assembly language routines, but it is important enough to warrant a mention here. Although the following discussion will center on keyboard filters, the same principles will hold true for drivers as well. Unlike filters for devices other than the keyboard, *KI filters MUST know whether or not JCL is active when installing themselves. To understand why, picture the chain of events that occur when a DO command is issued and a JCL file starts executing:

1) Pick up the current KI driver address from the DCB and save it away in the KIJCL$ storage area.

2) Check where a keyboard request comes from. If a LINE request, use a JCL file line; if a single KEY request, use the saved KI driver address to get it.

Step number 1 is the important point to understand. When DO is executing a JCL file, the driver address in the DCB is -NOT- the normal driver address; that address is temporarily stored away in the KIJCL$ area. Thus a filter program must do the following:

1) Check SFLAG$, bit 5, to see if DO is in effect. If it is not, proceed as normal and disregard steps 2 and 3.

2) Otherwise, pick up the real KI driver address from the KIJCL$ area for use by the filter program.

3) Store the new entry point for the KI device (taking into account the filter being installed) in the KIJCL$ area rather than in the DCB.

For those of you who have the 1st filter package, the source code for the CALC/FLT and the XLATE/FLT programs show actual examples of the test for an active DO during filter installation. Also, it is important that you disable the interrupts during the time the new address is actually being stuffed into the KI DCB area. It's not nice when someone presses a key when only 1 byte of the new address has been changed but not the other!

This month's column closes with a question. The first 3 people answering it correctly and have entries postmarked on the -correct date- will win copies of the FED, LED, the Utility disk, or one of the FILTER disks. (LSI employees and regular Quarterly contributors are excluded from this contest). Since this publication is mailed 3rd class, the -correct date- will be set at November 15th to give everyone a chance. To enter, write your solution on a postcard, in a letter, or on any other mail-able substance. In the event that more than 3 winning answers are received with the same postmark, a drawing will be held to determine the winners. Winners will be announced in next issue's JCL corner, along with the correct solution (the correct solution being the most fatal error that the following JCL file has IN MY OPINION).

JCL Question of the Quarter

QUESTION: What is the MOST fatal error of the following JCL example, and why:

```
//. This example will BACKUP a disk in drive ##1
//. To a disk in drive ##2
//. Tokens to be entered are:
//.
//. Enter the drive 1 disk as S= (source).
//. Enter the drive 2 disk as D= (destination).
//.
//. Evaluation of parameters starting . . .
//IF S
//. Source drive = #S#
//END if source drive
//IF D
//. Destination drive = #D#
//END if destination drive
//IF S&D
//. BACKING UP #S# to #D#
Backup :#S# :#D#
//END
```

This one is relatively easy. The next one will be a bit harder hopefully. If you can think of a good JCL puzzle, send it in in care of the LDOS Quarterly Editor. Until then, keep on DOing!

Roy's Technical Corner - 4Q82 - by Roy Soltoff

This issue's column will take a slightly different posture. In lieu of one major topic encompassing several pages of technical discussion, I will shed some light on the techniques for interfacing with a few system functions. This divergence is due in part to the tremendous amount of time spent during the past months in getting the LC compiler from MISOSYS ready for release. That and the work done to prepare LDOS 5.1.3 for Radio Shack has certainly impacted my time for authoring. I have also been busy getting the MSP-01 package released. This package has a few unique programs. If you ever use Job Control Language (JCL), you ought to look at the PARMDIR program which is part of MSP-01. The other major piece of MSP-01 is DOCONFIG - similar to Les's SYSGEN program; however, DOCONFIG functions while running JCL without interference.

There have been a few queries concerning the issues to be discussed in this column - so I really am NOT straying too far. I will address assembly language handling of BYTE I/O for error detection, interfacing to the @ICNFG vector, interfacing to the @KITSK vector, testing for 5.1.3 vs < 5.1.3 (i.e. 5.1.2, 5.1.1, 5.1.0), handling the KFLAG$ scanner for <BREAK> and <PAUSE> detection, and possibly some other items to look out for if you are doing some advanced coding.

Before I go any further in this issue's column, I must correct an error that appeared last time. The article on the Task Processor that appeared in the July 1982 issue needs a correction on page 32. The code that reads:

```
        LD      DE,MYTASK    should       LD      DE,MYTCB
        LD      A,2            be         LD      A,2
        CALL    @ADTSK                    CALL    @ADTSK
```

Let me begin this issue with error handling during byte I/O. LDOS has a great degree of device independence. However, due to incomplete device driver design considerations when the TRS-80 ROM was first done on the Model I (and further mishandled on the Model III), one big problem is inherent with byte I/O handling. Any device or file may be accessed for input or output on a byte I/O basis using the @GET and @PUT vectors (see "Device I/O and Independence" by Roy Soltoff in THE LDOS QUARTERLY, Volume I, Number 3).

If I/O through @GET/@PUT is coupled to a disk file, then the flag state on return is the Z-flag set if no error occurred while the Z-flag is reset if there was an error detected and the accumulator will contain the error return code. The byte I/O device drivers do NOT maintain this concept. Thus, if you @PUT to the *PR device, the state of the Z-flag is indeterminate. Also, if you @GET from the *KI device, the Z flag is indicative of a returned character.

LDOS permits a redirection of device I/O via the ROUTE command. LDOS also permits the OPEN routine to open a device or a file. Thus, if a device/file is opened or a device is routed, the application program invoking the byte I/O does not know if the physical I/O is from/to a device driver that does not use the Z-flag for error reporting or a disk file which does. This illustrates the age old problem of detecting a disk full condition when you have routed the *PR device to a file and proceed to "LPRINT" data. The application that is calling @PRT (the system vector to print a byte of data) is not expecting the output to go to a disk file and thus wouldn't normally be checking for errors. In fact, if it did check the Z-flag while @PRT was directed to a PRINTER, it would probably detect an error on every character depending on what state the Z-flag was left in.

This is unfortunate! When we get beyond the ROM-based LDOS, I will correct this deficiency by ensuring that all byte I/O handlers maintain the state of the Z-flag purely for error feedback. But what can you do now? Well, there is an interim solution to handle the error feedback if you place a particular routine within your assembler application. Let's look at the following portion of the "PUTOUT" routine.

```
PUTOUT  EQU     $
        LD      DE,FCBOUT       ;OUTPUT control block
        CALL    @PUT            ;To device/file
        RET     Z               ;Back if no "error"
```

We will assume that FCBOUT contains the control block data for some Output device/file. Any place in our program that we need to write a byte of data to this device/file, we will call PUTOUT. The first three instructions are very straightforward. The first loads the control block address into register pair DE. Next we call the @PUT routine which writes the byte that is in the accumulator register. So far, so good. Next we return to whatever called PUTOUT, but only if the Z-flag was set (i.e. no error during disk I/O). If the Z-flag was reset, we know that either we got a disk I/O error OR the output was to a byte device which does not properly maintain the Z-flag. How can we isolate one or the other?

Remembering that bit-7 of the control block is only a "one" if we have an open file, we will test that bit. Let's take a peek at some more of the PUTOUT routine:

```
        RET     Z               ;... from above
TSTDEV  EX      DE,HL           ;Ck if device or file
        BIT     7,(HL)          ;  Bit-7 is set in
        EX      DE,HL           ;  FCB's only!
        RET     Z               ;Ret if device
IOERR   OR      40H             ;Short message, abort
        JP      @ERROR          ;  or your error handler
```

By exchanging registers HL and DE, we momentarily transfer the control block address to register pair HL and use a BIT instruction to test whether bit-7 is a "one" or a "zero". Now if the control block is really for a byte device, bit-7 is reset and the test will SET the Z-flag. Therefore, a byte I/O device will be detected and we return to the calling routine with a "no error" indication - both done at the same time.

On the other hand, if the control block was for a disk file, bit-7 is set providing an "NZ" indication - which is the same result as a disk error. Thus, we fall through to the error routine.

I prefer to treat input slightly different because I would like to support a means of passing an end-of-file condition to the calling routine. In some of the system modules, a <BREAK> is used to denote EOF from the keyboard device. In the PARMDIR program, I chose to use <CONTROL-SLASH>. The reason for this is that <CONTROL-SLASH> generates an ASCII Field Separator (FS) which just happens to be X'1C' - the same as the EOF error code for disk files. Now let's look at my input routine.

```
GETIT   LD      DE,FCBIN        ;Input control block
        CALL    @GET
        RET     Z               ;Ret if no error
        CP      1CH             ;EOF?
        JR      NZ,TSTDEV       ;Ck device if not EOF
        OR      A               ;Set NZ flag
        RET
```

We are fetching the byte of data in the first two instructions in a normal manner. As with PUTOUT, we return if no error is detected. Here, I then look for an EOF error. If I detect some other error (or we are fetching from a byte device which misuses the Z-flag), then the routine jumps to the TSTDEV routine which makes the determination of device or file and acts accordingly.

If on the other hand, the EOF error code was in the accumulator, GETIT will return with the Z-flag reset. That means that we return to the calling routine always with the Z-flag SET if there has been a byte validly fetched; we return with the Z-flag reset and EOF error code on an end-of-file. Since the EOF error code is really useful to indicate when we are at the end of a file and generally does not represent an "error", to speak of, we have the routines that call GETIT check the Z-flag result for EOF indication.

With these routines, I can have a program obtain input based on a command-line or prompted input devspec/filespec. Then, if the devspec was used and the *KI device was referenced, a <CONTROL-SLASH> indicates the "end of the file". In PARMDIR, this exact set of routines is used to support parameter input from either a file (P="filespec") or the keyboard device (P="*KI").

@CKDRV 5.1.2/5.1.3 Handling

The next mini-topic will cover the method to determine if the machine is running LDOS 5.1.3 or 5.1.2. This is specifically important if you want to use the @CKDRV vector which changed in location on the Model III under release 5.1.3. Since OSVER$ contains X'51' under all releases of LDOS 5.1.x, it is necessary to ascertain which release a program is running under so that the proper @CKDRV address will be used.

All programs currently released by MISOSYS, determine the machine model (Model I or III) by checking the byte at address X'125'. Where vector references exist that are different on each machine model, the reference is updated at the start of the program. MISOSYS programs use the Model I vector inline and switch to the Model III vector if the machine check byte is indicative of a Model III (Note: not all of my programs have been updated to reflect this 5.1.3 test).

The following routine is abbreviated and only illustrates the @CKDRV vector modification.

```
@CKDRV1 EQU     44B8H           ;Model I all releases
@CKDRV3 EQU     4290H           ;Model III to 5.1.2
@CKDRVZ EQU     4209H           ;Model III 5.1.3
BEGIN   EQU     $               ;Start of my program
        PUSH    HL
        LD      A,(125H)        ;Model I or III
        CP      'I'             ;   III has 'I'
        JR      NZ,MODEL1
        LD      HL,@CKDRV3      ;Try "to 5.1.2" first
        LD      A,(@CKDRV3+1)   ;P/u handler SVC #
        CP      0C4H            ;Is it CKDRV or RAMDIR?
        JR      Z,$+4           ;Go if 5.1.2 or earlier
        LD      L,@CKDRVZ&0FFH  ;  else use 5.1.3's
        LD      (CKDRV+1),HL    ;Update vector within pgm
MODEL1  POP     HL
```

The routine is simplistic. The important test is the "CP 0C4H" instruction. Since the @CKDRV and RAMDIR vectors need to recover a system overlay request number, the contents of X'4291' will be different under 5.1.2 and 5.1.3 only on the Model III. This is the test for determination. The "L,@CKDRVZ&0FFH" is a minimal code method to update the low-order byte of a 16-bit register pair when the high order byte remains the same.

KFLAG Scanner interfacing

The KFLAG scanner was introduced in version 5.1.0. Since the scanner has not been documented too well, and rigorous methods of its use have not been disclosed, I thought I would reveal an example of its integration into an application.

A little background is in order. Many applications have the need to detect a PAUSE or BREAK condition while they are running. BASIC does this after every logical statement is executed (i.e. end of line or ":"). That's how, in BASIC, you can stop a program with the <BREAK> key or pause a listing. The classical way to detect the condition was to call @KBD checking for <BREAK> or <PAUSE> (SHIFT-@) ignoring all other keys. Unfortunately, if you were trying to make use of type-ahead, the @KBD call would flush out the type-ahead buffer if any keys were stacked up; thus, type ahead would be ineffective.

Another method actually used in 5.0.x was to scan the keyboard by physically examining the keyboard matrix. Again, a detrimental side effect resulted with type-ahead also storing up the keyboard depression for some future unexpected input request. Examining the keyboard directly also inhibits remote terminals from passing the <BREAK> or <PAUSE> condition.

The KFLAG scanner grew from these deficiencies. The scanner is part of the interrupt processor and examines the keyboard for three functions, <BREAK>, <PAUSE>, and <ENTER>. If any of theses conditions are detected, appropriate bits in the KFLAG$ are set (bits 0, 1, and 2 respectively). It is IMPORTANT to note that the interrupt KFLAG scanner does NOT reset the bits - it only sets them. Thus, it is up to the application using these flag conditions to reset the bits as required. Now, you may ask, why wasn't the scanner coded so that it resets the bits? Well now, if that was the case, you would never sense the "events" as they would occur too fast. Think of the KFLAG$ bits as a latch. With this little introduction, let's look at a routine I designed to use the <BREAK> and <PAUSE> conditions.

```
CKPAWS  LD      A,(KFLAG$)      ;P/u the flag
        RRCA                    ;Bit 0 to carry
        JP      C,GOTBRK        ;Go on BREAK
        RRCA                    ;Bit 1 to carry
        RET     NC              ;Return if no pause
        CALL    RESKFL          ;Reset the flag
        PUSH    DE
        LD      DE,KDCB$
        XOR     A
        CALL    @CTL            ;Flush the type-ahead
        POP     DE
PROMPT  PUSH    DE
        CALL    @KEY            ;  & wait for key entry
        POP     DE
        CP      1
        JP      Z,GOTBRK
        CP      60H
        JR      Z,PROMPT
RESKFL  PUSH    HL
        PUSH    AF
        LD      HL,(CKPAWS+l)   ;P/u KFLAG pointer
RESKFL1 LD      A,(HL)          ;P/u the flag
        AND     0F8H            ;Strip ENTER, PAUSE, BRK
RESKFL2 OR      40H             ;Set ECM
        LD      (HL),A
        PUSH    BC
        LD      B,16
        CALL    @PAUSE          ;Pause a bit
        POP     BC
        LD      A,(HL)          ;If got set again...
        AND     3
        JR      NZ,RESKFL1      ;  then reset it again
        POP     AF              ;Restore possible prompt
        POP     HL              ;  char & exit
        RET
```

I think that the best thing to do would be to take apart this entire routine and explain each sub-routine. The first piece:

```
CKPAWS  LD      A,(KFLAG$)      ;P/u the flag
        RRCA                    ;Bit 0 to carry
        JP      C,GOTBRK        ;Go on BREAK
        RRCA                    ;Bit 1 to carry
        RET     NC              ;Return if no pause
```

reads the KFLAG$ contents. The first rotate instruction places the BREAK bit into the carry flag. Thus, if a <BREAK> condition was in effect, the sub-routine would branch to "GOTBRK" - which is your break handling routine. If there is no pending BREAK, the second rotate places what was originally in the PAUSE bit into the carry flag. If no <PAUSE> condition is in effect, the routine returns to the caller. This sequence of code gives a higher priority to <BREAK> (i.e. if both BREAK and PAUSE conditions are pending, the <BREAK> condition has precedence). It is important to note that the GOTBRK routine needs to clear the KFLAG$ bits after it services the <BREAK> condition. This is simply done via a call to RESKFL.

The next part of the routine is executed on a <PAUSE> condition.

```
        CALL    RESKFL          ;Reset the flag
        PUSH    DE
        LD      DE,KDCB$
        XOR     A
        CALL    @CTL            ;Flush the type-ahead
        POP     DE
```

First the KFLAG$ bits are reset via the call to RESKFL. Next, we take care of the possibility that type-ahead is active. If it is, the PAUSE key was most likely detected by the type-ahead routine and thus the PAUSE is stacked there also. We want to flush (remove all stored characters) out the typeahead buffer. There are a few ways of doing this. We could repeatedly call @KBD until no characters were remaining. We could also use the undocumented scheme of writing a zero to the *KI device through the @CTL call (i.e. a @CTL-0 written to *KI clears the type-ahead buffer commencing with 5.1.2).

Now that we are in a PAUSEd state and the type-ahead buffer is cleared, we need to wait for a key input. The following routine does this:

```
PROMPT  PUSH    DE
        CALL    @KEY            ;  & wait for key entry
        POP     DE
        CP      1
        JP      Z,GOTBRK
        CP      60H
        JR      Z,PROMPT
```

The PROMPT routine will accept a <BREAK> and branch to your BREAK handling routine. It will ignore repeated <PAUSE> (the 60H). Any other character will cause it to fall through to the following routine which clears the KFLAG$.

```
RESKFL  PUSH    HL
        PUSH    AF
        LD      HL,(CKPAWS+l)   ;P/u KFLAG pointer
RESKFL1 LD      A,(HL)          ;P/u the flag
        AND     0F8H            ;Strip ENTER, PAUSE, BRK
RESKFL2 OR      40H             ;Set ECM
        LD      (HL),A
        PUSH    BC
        LD      B,16
        CALL    @PAUSE          ;Pause a bit
        POP     BC
```

```
        LD      A,(HL)          ;If got set again...
        AND     3
        JR      NZ,RESKFL1      ;  then reset it again
        POP     AF              ;Restore possible prompt
        POP     HL              ;  char & exit
        RET
```

The RESKFL subroutine needs to be called when you first enter your application. This is necessary to clear the flag bits that were probably in a "set" condition. This "primes" the detection. The routine also needs to be called once a BREAK, PAUSE, or ENTER condition is detected and handled (its only necessary to deal with the flag bits for the conditions you are using).

Notice that throughout the entire CKPAWS routine, "KFLAG$" was referred to in only one instruction. This was done so that the Model check (I or III) need only update one address in memory. Now you can clean-up your "paws".

Interfacing to @ICNFG

Many years ago in a galaxy far far away... Actually, it's probably been between one and two years ago. With the capability of SYSGEN, many of our users preferred to SYSGEN the RS-232 driver. When that was attempted, it worked okay as long as you did not power down your machine. This particular limitation was certainly something we all could not live with. The problem was that the RS-232 hardware (UART, Baud Rate Generator, etc.) needs to be initialized before it can be used. It wasn't good enough to just configure with the RS-232 driver resident, some initialization routine was necessary. Thus, the need to be able to invoke, at BOOT, a routine to initialize the RS-232 driver, became evident. Out of this came the @ICNFG vector (which is an acronym for "initialization configuration").

The @ICNFG vector is always called by the SYS0 initialization stub after any configuration file is loaded. Thus, any initialization routine that is part of a high-memory configuration, can be invoked by chaining into @ICNFG. The following procedure may be examined to illustrate this link. The first thing to do is to move the contents of the @ICNFG vector into your initialization routine. The subroutine:

```
        LD      A,(@ICNFG)              ;Get opcode
        LD      (LINK),A
        LD      HL,(@ICNFG+1)           ;Get address
        LD      (LINK+1),HL
```

does this by transferring the three byte vector to your routine. You then need to relocate your routine to its execution memory address. Once this is done, transfer the initialization entry point to the @ICNFG vector as a jump instruction with:

```
        LD      HL,INIT                 ;Get (relocated)
        LD      (@ICNFG+1),HL           ;  init address
        LD      A,0C3H                  ;Set JP instruction
        LD      (@ICNFG),A
```

If you need to invoke the initialization routine at this point, then you can:

```
        CALL    @ICNFG                  ;Initialize routine
```

Your initialization routine would obviously be unique to the function it was to perform. A template for such a routine would appear as:

```
INIT    EQU     $                       ;Start of mit
        .
        your initialization routine
        .
LINK    DB      'LSI'                   ;continue on
```

Don't forget to SYSGEN after linking in your routine. By following these procedures, you can effect the invocation of your routine every time you boot LDOS.

Interfacing to @KITSK

Consider for a moment that disk I/O can not take place during an interrupt task. How then can we write "background" routines that perform disk I/O? The system printer spooler does its despooling function as a background task. If we cannot perform disk I/O during interrupt tasks, how can we despool? We achieve this by being able to invoke background tasks in one of two ways. We can use the RTC interrupt (or other external interrupt). Thus, type cannot be used to perform disk I/O. We can also use the keyboard task.

At the beginning of the LDOS keyboard driver is a call to @KITSK. This means that any time @KBD is called, the @KITSK vector is likewise called (actually, the type-ahead interrupt task bypasses this entry to inhibit calling @KITSK from the interrupt routine). Therefore, if you want to interface a background routine that does disk I/O, you must chain into @KITSK

The interfacing procedure to @KITSK is virtually identical to that shown above for @ICNFG and will not be repeated here. For the sake of clarity, you may want to write your background routine to start with:

```
START   CALL    ROUTINE         ;Invoke task
LINK    DB      'LSI'           ;Space for KITSK hook
ROUTINE EQU     $               ;Start of the task
```

Now that I have demonstrated the procedure) let me point out one major pitfall. The @KBD routine is invoked from @CMD which is in SYS1. This invocation is from the @KEYIN call which fetches the next command line after issuing the "LDOS Ready" message. If your background task executes and opens or closes a file (or does anything to cause the execution of a system overlay other than SYS1), then SYS1 will be overwritten by SYS2 or SYS3 respectively). When your routine finishes, the @KEYIN handler returns to what called it which was SYS1. Unfortunately, SYS1 is no longer resident. Thus crash city is upon you.

ANY TASK CHAINED TO @KITSK WHICH CAUSES
A RESIDENT SYS1 TO BE OVERWRITTEN MUST
RELOAD SYS1 PRIOR TO RETURNING.

Okay, how do you accomplish this without knowing system code (point of information: if you are writing background tasks, you are writing system support code!)? You will be able to use the following code to reload SYS1 if SYS1 was resident prior to your task's execution. Don't forget to correct any Model I/III vectors if your code is to run on either machine.

```
ROUTINE LD      A,(OVRLY$)      ;P/u resident overlay
        AND     8FH             ;  and remove entry
        LD      (OLDSYS+1),A
        .
        Rest of your task
        .
EXIT    EQU     $
OLDSYS  LD      A,0             ;P/u old overlay$
        CP      83H             ;Was it SYS1?
        CALL    Z,GETSYS1       ;Get sys1 back if it was
        RET
GETSYS1 RST     28H             ;Fetch SYS per reg A
```

Miscellaneous Tidbits

If you have a program that relocates to high memory, here is a caution to observe. Note that any command can be executed from LBASIC via the <CMD"command string>. LBASIC accomplishes this by shifting the BASIC program and variables to available high memory and lowering HIGH$. That's why we inhibit the execution of certain library commands that effect changes to HIGH$ (ever see the "Can't - Valid only at LDOS" mesage?). Your program should do likewise. Therefore, if your program behaves in the above manner, have it check the contents of @EXIT. If the @CMD request came from LBASIC, then @EXIT will contain the start of a jump instruction (X'C3'). Thus, if it's a C3H, then inhibit your program from executing. If it's not a C3H, your program is okay for execution.

Well, that's it for this issue. If you have specific requests for the next issue's "Roy's Technical Corner", drop a line to me at LSI. Good coding!

LES INFORMATION

by Les Mikesell

This column will deal with the technical aspects of the LDOS RS232 drivers and some general information about using device I/O from within programs. The principle is quite simple: any program can substitute a device name for a filespec in the routine to OPEN a file, then use calls to @GET (013H) or @PUT (01BH) to input or output characters through the device. OPENing a device actually creates a device control block that is ROUTEd to the DCB that is known to the system. An LBASIC program can use this function to OUTPUT to a device (such as *CL) by using statement OPEN "O",1,"*CL", followed by PRINT#1,"string", where "string" can be one or more characters. As in the sequential file access mode, a semi-colon must be used after the string if a carriage return should not automatically be added at the end of the string. INPUT from devices is also allowed, but is somewhat restricted by the lack of a single character input (corresponding to INKEY$) that is usable with files or devices. Also, INPUT and LINEINPUT require a linefeed following the carriage return to terminate the input when used with devices, due to the method that is used to skip the linefeeds if encountered when the input is done from a file. This problem will be addressed later. First, a look at how it is done in machine code. The following program could be used as a dumb terminal or to test input and output with any device. It is used by passing the device name on the command line (like LCOMM).

```
; Simple terminal program using RS232/DVR
;
@KBD    EQU     02BH    ; get key press
@DSP    EQU     033H    ; display single character
@DSPLY  EQU     4467H   ; display line
@FSPEC  EQU     441CH   ; move file/device name
@OPEN   EQU     4424H   ; open existing file/device
@ERROR  EQU     4409H   ; get system error message
@GET    EQU     013H    ; input character fm fileldevice
@PUT    EQU     01BH    ; output character to file/device
@EXIT   EQU     402DH   ; normal exit to LDOS
@ABORT  EQU     4030H   ; error exit
;
        ORG     5200H
;program is entered with HL pointing one space past file name
START:  PUSH    HL             ;save pointer to command line
        LD      HL,LOGON       ;point to log on message
        CALL    @DSPLY         ;display message
        POP     HL             ;restore command line pointer
        LD      A,(HL)         ;check character there
        CP      '*'            ;is it an * ?
```

```
        JP      NZ,NOTDEV       ;if not, quit!
        LD      DE,DCB          ;try to OPEN it
        CALL    @FSPEC          ;first move name to DCB
        JP      NZ,NOTDEV       ;quit if error
        CALL    @OPEN           ;open device now
        JR      Z,TERMIN        ;all's well..
        OR      0C0H            ;else set bits 6 & 7 of error
        CALL    @ERROR          ;get system error message
        JP      NOTDEV          ;then ours, and quit
;
TERMIN: LD      DE,DCB          ;point to device
TERM2:  CALL    @GET            ;test for input from R5232
        JR      Z,KEYCHK        ;none, try keyboard
        CALL    @DSP            ;display received character
KEYCHK: CALL    @KBD            ;check for keypress
        JR      Z,TERMIN        ;no key press, try R5232
        CP      1               ; "BREAK" pressed?
        JP      Z,@EXIT         ;back to LDOS if so..
        LD      DE,DCB          ;point to R5232 device
        CALL    @PUT            ;send character
        JP      TERM2           ;continue till break pressed
;
DCB:    DS      32              ;space for device control block
;
NOTDEV: LD      HL,ERRMSG       ;point to error message
        CALL    @DSPLY          ;display it
        JP      @ABORT          ;quit
;
ERRMSG: DEFM    'Device Spec Required!'
        DB      0DH             ; carriage return
;
LOGON:  DEFM    'LDOS dumb terminal program'
        DB      0AH             ; line feed
        DB      0DH             ; carriage return
 ;
        END     START
```

The program can, of course be expanded to include any additional functions that are desired. It demonstrates the use of @GET and @PUT, but the device drivers may also allow using the @CTL call (023H) to pass various control information between the program and driver. The control functions are accessed by loading DE with the address of the DCB, A with the control function number, and calling @CTL. If the driver returns a value to the program, it will be passed in the A register or the status flag register. The functions currently provided by the LDOS drivers are:

With the printer and RS232 drivers, sending a zero through @CTL will cause the driver to test the output status of the device (can it accept a character now?) and set the Z flag accordingly (Z set means READY). This is the method used by the LDOS spooler to quickly determine if it can output any characters during an interrupt cycle.

KI/DVR accepts a zero through @CTL as a command to clear out the type-ahead buffer. (5.1.2 and 5.1.3)

The RS232T/DVR (for the Model 3) also includes two additional functions that are related to its abiltity to receive and buffer characters using the hardware interrupt from the RS232 interface. Passing a "W" (57H) through @CTL tells the driver that an address is being passed in the IY register. This will be installed as a CALL address in the interrupt processing routine. After each character is received and put in the RS232T buffer, a CALL will be made to the specified address (the "wakeup" function).

The subroutine at that address would typically call @GET to get the input character and store it in a dedicated buffer for the program (Maintaining the buffer pointers can be tricky, since characters can be added at any time). Passing an ASCII X (58H) through @CTL will disable the CALL set up by the "W" function, and obviously must be used before exiting from any program that has used "wakeup".

Now, for those who prefer to program in BASIC, I will show a couple of ways to access the @GET routine to input from a device within a BASIC program. The most common need for this ability would be to input characters from the RS232, but the principles would apply to any device that can supply input.

The first method is a quick and dirty approach that only requires a few PEEKs and POKEs to temporarily ROUTE the keyboard input requests to the desired device driver, so that INKEY$, INPUT or LINEINPUT statements may be used to access the device. It may only be used in the Model 1, due to a difference in the way ROUTEs are handled in the Model 3. First, set up the *SI device by SETing it to the appropriate driver at LDOS command level (e.g. SET *SI to RS232R), or ROUTE it to the desired device if it already exists in your system (e.g. ROUTE *SI to *CL). Note that the ROUTE command for an input device seems backwards - it actually means "ROUTE the input requests for <source device> to <destination device>". The important point here is to get one of the "known" extra DCBs set up for input from the desired device. This configuration may be saved with SYSTEM (SYSGEN). Since the *KI and *SI DCBs are in fixed memory locations, a BASIC program can use PEEKs and POKEs to simulate the ROUTE command at any time. Now, PEEK the first three bytes from the *KI DCB (4015H - 4017H) and store the values, then POKE a 16 (hex 10 - to indicate a ROUTE) into 4015H, followed by the address of the *SI DCB in least significant, most significant format (this is different on the Model 3, so this technique cannot be used). For the Model 1, this would be POKE (&H4016),(&HC8): POKE (&H4017),(&H43) to set up the keyboard device control block in a ROUTEd condition, so that any input requests that would normally go to the keyboard driver will temporarily be satisfied by the *SI device instead. To restore the normal keyboard, simply POKE back the saved original values. If type-ahead is active, any keys that were pressed while the DCB was ROUTEd will still be available when the DCB is restored to the KI/DVR.

A few precautions must be taken to make this method safe. save the current *KI DCB contents EVERY TIME before POKEing in the ROUTE. Then you can be sure that you are restoring the correct driver when you POKE these values back. Remember that JCL also uses the *KI DCB, and if the /JCL file terminates in the BASIC program, the contents of the DCB will be altered. This can only happen when the "real" keyboard is enabled, so saving/restoring the values each time will avoid most problems. Also, the "real" keyboard driver should be active whenever possible and the BREAK key locked out while the *KI is ROUTEd. The system BREAK bit is set during interrupt processing, and it would be possible to break a BASIC program with NO access to the keyboard if the ROUTE is still in place, or even worse, the break could occur in the middle of POKEing the DCB, which would crash the system!

Sample LBASIC terminal program for Model 1 ONLY
   at LDOS Ready, SET *SI R5232R

```
5 CLEAR 5000: DIM I%(3)
10 OPEN "O",1,"*SI"  'same device can be used for output
20 GOSUB 100         'get input from KB
30 IF IN$<>"" THEN PRINT#1,IN$; 'send any key pressed
40 GOSUB 200         'ROUTE input to *SI
50 GOSUB 100         'now input routine will check RS232
60 GOSUB 300         'restore keyboard
70 IF IN$<>"" THEN PRINT IN$; 'display anything received
80 GOTO 20           'continue forever
90 ' subroutines
100 IN$=INKEY$:RETURN 'note there is no wait for input here
200 FOR I%=0 TO 2           ' get three bytes
```

```
210 KI%(I%)=PEEK(&H4015+I%) 'from the *KI DCB
220 NEXT : CMD"B","OFF"     'can't BREAK now!
230 POKE (&H4015),(&H10)    'indicate ROUTE
240 POKE (&H4016),(&HC8):POKE (&H4017),(&H42) '<=43 for Mod 1
250 RETURN
300 FOR I%=0 TO 2            'three bytes
310 POKE (&H4015+I%),KI%(I%) 'restore previous KB driver
320 NEXT: CMD"B","ON"        'OK to BREAK again
330 RETURN
```

This program would work at 110 baud or less as a dumb terminal, or the technique could be used where there is some form of handshaking to prevent dropped characters. The restrictions on this type of device access are: 1) it is slow if the swap is done for every character, 2) it is not possible to input a zero character value, so it would not work for binary data transfers where the null character is valid, 3) one of the "known" spare DCBs must be used for the alternate input device, and 4) it only works in the Model 1.

A USR routine called from BASIC can avoid these problems, but this approach is slightly more complicated, since the machine language routine must be installed. It is still possible to let BASIC do most of the work, by OPENing the device for output to create the device control block and allocate some buffer space that can be used for the USR code. First OPEN the device in the "random" mode and FIELD a dummy string variable so the VARPTR value can be used to locate the buffer address. Then CLOSE the device and re-open for sequential output using the same buffer number. Random access to a device is not allowed, but the dummy FIELD allows the BASIC program to find the file buffer, which will remain in the same location when the file is re-opened for output. The DCB will be 32 bytes lower in memory. Then, a machine code routine can be POKEd into the buffer space to load DE with the DCB, CALL @GET, and return the value to BASIC as an integer number. Since integers are two byte values, the value in excess of 255 can be used as an indication of whether a valid character was available or not, so that all possible character values including 0 can be detected.

```
10 CLEAR 5000:DEFINT A-Z:GOSUB 120 'initialize I/O
20 'put loop first for speed - also delete remarks
30 'sample terminal program
40 IN$=INKEY$ 'Check keyboard
50 IF IN$<>"" THEN PRINT#1,IN$; 'send any characters
60 IN = USR1(1) 'check device input - the (1) is a dummy
70 IF IN THEN PRINT CHR$(IN-256); 'print rcvd chars
80 GOTO 40    'do it forever...
90 '
100 '
110 'initialization
120 OPEN "R",1,"*CL' 'this method can use any device name
130 FIELD 1,1 AS DU$ 'a dummy string just to find address
140 VP=VARPTR (DU$) 'String pointer address
150 BF=PEEK(VP+1)+256*PEEK(VP+2)'BF is address of file buffer
160 CLOSE 1 'end random mode
170 OPEN "O",1,"*CL" ' open again for sequential output
180 DCB=BF-32 : LB=DCB AND (&H00FF) 'low byte of DCB address
190 HB=(DCB AND (&HFF00))/256 'high byte of DCB address
200 POKE BF,(&H11)    'LD DE,... (start of USR code)
210 POKE BF+1,LB      'low byte of DCB address
220 POKE BF+2,HB      'high byte of DCB
230 POKE BF+3,(&HCD)  'CALL...
240 POKE BF+4,(&H13): POKE BF+5,0 ' @GET (13H)
250 POKE BF+6,(&H21)  'LD HL,...
260 POKE BF+7,0 : POKE BF+8,0 ' 0000
270 POKE BF+9,(&H28)   'JR Z,...
```

```
280 POKE BF+10,(&H02)  ' $+2 (return zero if no character)
290 POKE BF+11,(&H24)  'else INC H (set H=01)
300 POKE BF+12,(&H6F)  'LD L,A  (put character in L)
310 POKE BF+13,(&HC3)  'JP... re-enter BASIC and pass value
320 POKE BF+14,(&H9A): POKE BF+15,(&H0A) ' 0A9A address
340 '..... any other initialization, etc.....
350 DEFUSR1=BF 'USR routine entry point
360 RETURN
```

This method may still be too slow for dependable 300 baud operation but should keep up on a model 3. Remember that string handling is slow in BASIC and there is a danger of losing data if string space must be compressed (BASIC's infamous "garbage collection"). It could receive MUCH faster if blocks of data can be received and stored in an integer array, then unpacked and stored between blocks. The value returned by the USR function will be 0 if no character has come in, otherwise it will be 256 + the character value. This allows detecting a zero as a valid character when it is received. If the sending device is using parity, it will be necessary to AND 127 with the number to accept only the valid seven bits. After the machine code routine is POKEd into the file buffer, it is important not to CLOSE the device until the program is done with the USR function since that would clear the buffer and DCB. This would occur if the program contains a global CLOSE statement.

The following CLFLT/FLT is a general purpose communications filter. It includes features to add nulls after carriage returns, a delay between characters, and a linefeed after every carriage return. It also provides a mask parameter to remove parity bits during ASCII file reception.

```
00100 ;Communications filter for LDOS RS232 drivers to provide
00110 ;testing for modem carrier, delay between characters,
00120 ;and linefeeds and nulls after carriage returns.
00130 ; Enter with parameters:
00140 ; CARRIER = ON or Y  /default is OFF
00150 ;       Setting ON will cause all input or output requests
00160 ;       to be ignored unless the modem is receiving a
00170 ;       carrier signal
00180 ; ADDLF = ON or Y   /default is OFF
00190 ;       Setting ON will add a linefeed after each carriage
00200 ;       return.
00210 ; NULLS = 0 to 256  /default is 0
00220 ;       Number of nulls to send after each carriage return
00230 ; DELAY = 0 to X'FFFF' /default is 0
00240 ;       Variable timing delay between output characters to
00250 ;       allow sending to systems that cannot accept full
00260 ;       speed transmission.
00270 ; MASK = ON or OFF Idefault is OFF
00280 ;       Setting ON will strip the high bit from received
00290 ;       characters, removing parity bits that may have been
00300 ;       added by the sender.  Use only for transmissions in
00310 ;       the ASCII character range, not for 8-bit binary data.
00320 ; All parameters may be abbreviated with the first letter
00330 ;
00340 ; Hardware dependant EQUates.. Mod 1 addresses used
00350 RSHACK  EQU     -1      ;a logical TRUE for assembly
00360 LX80    EQU     0       ;logical FALSE (reverse for LX80)
00370 HIGH$   EQU     4411H   ;<=change to 4049H for Mod 1
00380 @PARAM  EQU     4454H   ;<= 4476H for Mod 1
00390 @LOGOT  EQU     428AH   ;<= 447BH for Mod 1
00400 ;
00410 ; General EQUates...
00420 @EXIT   EQU     402DH
```

```
00430 @ABORT  EQU     4030H
00440 @DSPLY  EQU     446/H
00450 @DELAY  EQU     60H
00460 LF      EQU     10      ;linefeed character
00470 CR      EQU     13      ;carriage return
00480 ;
00490 ; LDOS 'FILTER' command handler
00500 ;
00510         ORG     5200H
00520 ENTRY:  PUSH    DE               ;save DCB pointer
00530         POP     IX               ;into IX register
00540         PUSH    HL               ;save cmd line pointer
00550         LD      A,(DE)           ;Pick up DCB type byte
00560         PUSH    AF
00570         LD      HL,SIGNON        ;=>Signon message
00580         CALL    @DSPLY           ;print it
00590         POP     AF               ;restore type byte
00600         BIT     3,A              ;Device routed to NIL?
00610         JP      NZ,ISNIL         ;Go if so
00620         BIT     4,A              ;Routed?
00630         JP      NZ,ROUTED        ;Go if so (error)
00640         AND     7                ;does driver handle I/O
00650         CP      7                ; and @CTL?
00660         JP      NZ,DEVERR        ;Go if not
00670         POP     HL               ;restore cmd line pointer
00680         LD      DE,PRMTBL        ;Scan parameters
00690         CALL    @PARAM
00700         JP      NZ,PRMERR        ;quit if error
00710 ;
00720 ;test parameter values and initialize filter
00730         LD      BC,$-$            ;value set by @PARAM call
00740 ADDLF   EQU     $-2               ;<=here
00750         LD      A,C
00760         OR      B                 ;set flag
00770         LD      A,LF              ;load a line feed
00780         JR      Z,NXTST           ;go if not specified
00790         LD      (LFFLG),A         ;stuff byte if wanted
00800 ;
00810 NXTST:  LD      BC,$-$
00820 CARRY   EQU     $-2               ;<=setting for CARRIER
00830         LD      A,B
00840         OR      C
00850         JR      Z,CKMSK           ;no checking if zero
00860         LD      A,0FFH            ;Stuff FFH if check wanted
00870         LD      (CFLAG),A         ;for input request
00880 ;
00890 CKMSK:  LD      BC,$-$            ;MASK param
00900 MASK    EQU     $-2               ;set by @PARAM
00910         LD      A,B
00920         OR      C                 ;zero?
00930         JR      Z,GETDVR          ; no masking wanted
00940         XOR     A                 ; set zero
00950         LD      (MSK),A           ; set NOP instd of RET
00960 ;
00970 GETDVR: LD      H,(IX+2)          ;pull driver address from
00980         LD      L,(IX+1)          ;DCB of device
00990         LD      (DVRADD),HL       ;put where needed in filter
01000         LD      (DVR2),HL
01010         LD      (DVR3),HL
01020         LD      (DVR4),HL
01030         LO      HL,(HIGH$)        ;find top of available memory
01040         LD      (OLDMEM),HL       ;save in filter header
```

```
01050          PUSH    HL                 ;save
01060          LD      BC,LAST            ;end of relocated code
01070          PUSH    BC                 ;save
01080          XOR     A                  ;clear carry
01090          SBC     HL,BC              ;find offset of move
01100          EX      DE,HL              ;put into DE
01110          LD      HL,(REL1)          ;relocate absolute memory
01120          ADD     HL,DE              ; references used in the
01130          LD      (REL1),HL          ;moved code...
01140          LD      HL,(REL2)          ;by..
01150          ADD     HL,DE              ;adding offset of move
01160          LD      (REL2),HL
01170          LD      HL,(REL3)
01180          ADD     HL,DE
01190          LD      (REL3),HL
01200          LD      HL,(REL4)
01210          ADD     HL,DE
01220          LD      (REL4),HL
01230          LD      HL,(REL5)
01240          ADD     HL,DE
01250          LD      (REL5),HL
01260          POP     HL                 ;end of filter (now)
01270          POP     DE                 ;old HIGH$
01280          LD      BC,LAST-FENTRY+1 ;length of relocated code
01290          LDDR                       ;move it
01300          LD      (HIGH$),DE         ;set new HIGH0
01310          INC     DE                 ;point to filter entry point
01320          LD      (IX+1),E           ;Shove it in the DCB
01330          LD      (IX+2),D
01340 ;*=*=*
01350 EXIT:    JP      @EXIT              ;Done
01360 ;*=*=*
01370 ;        Error handling
01380 ;*=*=*
01390 ISNIL:   LD      HL,ISNIL$
01400          JR      ERROUT
01410 DEVERR: LD       HL,DEVER$
01420          JR      ERROUT
01430 ROUTED: LD       HL,ROUTD$
01440          JR      ERROUT
01450 PRMERR: LD       HL,PRMER$         ;'Parameter error'
01460 ERROUT: CALL     @LOGOT            ;Display and log
01470          JP      @ABORT            ;Quit
01480 ;*=*=*
01490 ;        Data area
01500 ;*=*=*
01510 SIGNON: DB 'CL/FLT - LDOS communications Filter'
01520          DB LF,CR
01530 PRMER$: DB 'Parameter error!',CR
01540 ISNIL$: DB 'Device not active!',CR
01550 DEVER$: DB 'Incorrect device type!',CR
01560 ROUTD$: DB 'Device is routed!',CR
01570 ;
01580 PRMTBL: DB 'ADDLF '
01590          DW  ADDLF
01600          DB 'A     '
01610          DW  ADDLF
01620          DB 'CARRIE'
01630          DW  CARRY
01640          DB 'C     '
01650          DW  CARRY
01660          DB 'DELAY '
```

```
01670         DW  DELAY
01680         DB 'D     '
01690         DW  DELAY
01700         DB 'NULLS '
01710         DW  NULLS
01720         DB 'N     '
01730         DW  NULLS
01740         DB 'MASK  '
01750         DW  MASK
01760         DB 'M     '
01770         DW  MASK
01780         DW      0       ;end of list
01790 ;
01800 ;*=*=*
01810 ;       Actual filter moved to high memory
01820 ;       LDOS style header...
01830 ;*=*=*
01840 FENTRY: JR      START           ;Branch around linkage
01850         DW      $-$             ;Last byte used
01860 OLDMEM  EQU     $-2             ;<=previous HIGH$ value
01870 ;
01880         DB      5,'CLFLT'
01890 ;
01900 ; actual filter routine
01910 ; the initialization code sets the flag byte to FFH if
01920 ; the test for carrier is wanted, 0 if not
01930 START:  LD      A,$-$    ;get flag for carrier test
01940 CFLAG   EQU     $-1      ;set according to params
01950         JR      C,INPUT  ;an input request
01960         JR      Z,OUTPUT ;output request from program
01970 ;  fall through if program called @CTL
01980 GSTAT:  OR      0FFH    ;reset C and Z so filter can...
01990                         ;call driver for status
02000 DRIVER: JP      $-$     ;go to old driver
02010 DVRADD  EQU     $-2     ;stuff driver address here
02020 ;
02030 ;
02040 ; use a test that will set NZ if going directly to the driver
02050 INPUT:  INC     A         ;set Z if flag was FF for carrier
02060 ;
02070         CALL    Z,STAT    ;check for carrier if wanted
02080 REL1    EQU     $-2       ;call address is relocated
02090         JR      Z,IGNORE  ; no carrier, skip it
02100         SCF               ; input wanted
02110         CALL    $-$       ;driver address
02120 DVR2    EQU     $-2       ;stuffed by loader
02130 MSK:    RET               ;replaced w/NOP if MASK specified
02140         AND  7FH          ;strip parity bit
02150         RET
02160 ;
02170 ;  no carrier, so call driver to clear UART and buffer
02180 ;  then ignore any received characters
02190 ;
02200 IGNORE: OR      0FFH    ;be sure Z flag is off
02210         SCF             ;carry flag on
02220         CALL    $-$     ;get input from driver
02230 DVR3    EQU     $-2     ;driver address
02240         XOR     A       ;throw character away
02250         RET
02260 ;
02270 OUTPUT: PUSH    BC      ;save @PUT character
02280         INC     A       ;Test flag (zero to skip)
```

```
02290          CALL    Z,STAT  ;test carrier if flag was FF
02300 REL2     EQU     $-2
02310          POP     BC       ;restore character
02320          JR      Z,FAKEIT ;no carrier, dump output character
02330 ;
02340          PUSH    BC       ;save character
02350          CALL    SLOW     ;delay/send
02360 REL3     EQU     $-2      ;address is relocated
02370          POP     BC
02380          LD      A,CR      ;check for carriage return
02390          CP      C         ;being output
02400          JR      NZ,FAKEIT ;done if not
02410 ;
02420          LD      A,$-$     ;A 0 or LF (set when loaded)
02430 LFFLG    EQU     $-1     ;stuff byte according to parameter
02440          LD      DE,$-$  ;pick up NULL count
02450 NULLS    EQU     $-2     ;set by @PARAM
02460          PUSH    DE      ;save count
02470          LD      C,A     ;the @PUT char (0 or LF)
02480          OR      A       ;LF needed?
02490          JR      NZ,SEND ;output if so
02500 ;
02510          POP     DE      ;else restore NULL count
02520 NLOOP:   LD      A,D
02530          OR      E       ;test if done
02540          JR      Z,FINAL ;go if finished
02550          DEC     DE      ;else count down nulls
02560          PUSH    DE      ;save count
02570 SEND:    CALL    SLOW    ;pause/send character
02580 REL4     EQU     $-2     ;address goes here
02590          LD      C,0     ;load null to send
02600          POP     DE      ;restore count of nulls wanted
02610          JR      NLOOP   ;send until done
02620 ;
02630 SLOW:    PUSH    BC       ;save @PUT char
02640          LD      BC,$-$   ;get DELAY value (loaded by @PARAM)
02650 DELAY    EQU     $-2      ;value stuffed by @PARAM
02660          LD      A,B
02670          OR      C        ;check for zero (default)
02680          CALL    NZ,@DELAY  ;(always sets Z flag)
02690          POP     BC       ;restore character
02700          CALL    $-$      ;driver, to output character
02710 DVR4     EQU     $-2
02720 ;
02730 FINAL:   LD      A,CR    ;Load the original @PUT char
02740          LD      C,A     ;into C and A in case other
02750                          ;filters are active
02760 FAKEIT:  CP      A       ;set Z flag for good return
02770          RET             ;done with output
02780 ;
02790 ;
02800 STAT:    LD      C,0     ;for status check
02810          CALL    GSTAT   ;call driver for status
02820 REL5     EQU     $-2     ;relocated when loaded
02830 ;
02840 ;the status call returns the modem status byte from the UART
02850 ;or SIO chip in register A, as well as the output status
02860 ;in the Z flag, but the bit values depend on the hardware
02870 ;
02880          IF RSHACK       ; this test applies to std Mod 1 or 3
02890          CPL             ; 0=ON in UART, so flip bits
02900          BIT     5,A     ;check for carrier signal bit
```

```
02910          ENDIF
02920 ;
02930          IF LX80
02940          NOP             ;patch space to modify
02950          BIT     3,A     ;carrier bit from LX80 SIO
02960          ENDIF
02970 ; Z=no carrier, NZ=carrier on...
02980          RET
02990 LAST     EQU     $-1  ;used for length calculation
03000 ;
03010          END     ENTRY
```

LATE BREAKING NEWS AND OTHER ASSORTED ITEMS

The following is a series of patches for Radio Shack's new SuperScripsit. They were done by Tom Price and the people at Powersoft. Any questions on this article should be addressed to them.

Patching SUPER SCRIPSIT(tm) For Use With The LDOS 5.1.3 Operating System

By: Dennis A. Brent and Renato Reyes Ph.D

Radio Shack's new entry in the word processing jungle is out and it sure looks like a winner! SUPER SCRIPSIT is everything that Tandy has been promising and more. It incorporates many features which have been lacking in other WP programs, notably the ability to deal with files larger than available memory, and the ability to interface to a wide variety of printers. This new system has so much flexibility and so many neat features, that we will save the review for another article. Take our word, though, it is some of the nicest software to come out of Texas, outside of SUPER UTILITY!

Wouldn't it be nice to combine all this new word processing power, and use it under LDOS too! That is what we thought, and we decided to do something about it. On top of modifying SUPER SCRIPSIT for use with LDOS, we wanted to run SuperScripsit on our Laredo Hard Drive. All of the above is now possible, and has been well-tested. This information is available below, and also in the PowerSOFT XTRA-80 Sig and LDOS Sig Database on MicroNET for your down-loading pleasure. We do not have the new Radio Shack Hard Disk yet, but we assume that these patches will work just fine on THAT system running under LDOS 5.1.3. We have included a JCL file for easy application.

We also have developed printer drivers for the EPSON MX-series printers equipped with GRAFTRAX-80 or GRAFTRAX-Plus. These drivers interface directly with SuperScripsit and allow you to make use of the many printing modes and special features of your EPSON printer. The driver programs will permit you to print properly-justified text in any one of the four print sizes the EPSON is capable of: expanded print, double-wide compressed print, standard 10-chars.-per-inch print, and compressed print. The print sizes are selected by supplying the correct pitch value to SuperScripsit at document OPEN time.

All EPSON drivers support underlining, boldface, and italics using SuperScripsit control codes. In addition the Graftrax-Plus drivers (two are supplied) support superscripting and subscripting. When emphasized print mode is switched in, the drivers will automatically switch it off when printing superscripts and subscripts (otherwise the EPSON would refuse to do super/subscripts). The Graftrax-Plus drivers will also trap illegal conditions such as switching emphasized print mode on while in compressed print size (the GRAFTRAX-Plus ROMs sometimes respond to this by switching compressed print OFF, which is undesirable).

The EPSON printer drivers are available on disk for $29.95. Ask for the PowerSOFT drivers for SuperScript(tm). Included as a free bonus are all the patch files necessary to get SuperScripsit running under LDOS 5.1.3 on your disk. This way, you do not have to type them in!

We will develop SuperScripsit printer driver software for other printers and release them in the near future. In the works are a printer driver for the excellent C. Itoh PROWRITER which will support proportional printing, as well as a driver for the new ANADEX series of printers. We would like to thank Tom Price for his invaluable help and guidance in getting these patches in the "bug-free" state that they are in. We would also like to thank Earle Robinson for his testing and suggestions.

8/26/82

These are the patches necessary to make SuperScripsit from Radio Shack work correctly under LDOS 5.1.3. These patches permit the creation and maintenance of files which are an even multiple of 256K (B-I-G files!).

Model III

```
.SCRIPIII/FIX
.LDOS Patch to SuperScripsit by Tom Price and Renato Reyes
.Permits operation under LDOS 5.1.3 Mod III
.This patch compliments of PowerSOFT
.
.PATCH SCRIPSIT/CMD USING SCRIPIII/FIX
.
D00,08=3A 89 42 32 20 5B CD 01 5B
D05,2E=C3 16 5B
D09,0D=E5 CD 90
D09,10=42 E1 C0 23 23 46 23 7E B7 3E FF 20 03 00 00 78
D09,29=CB 8F 32 89 42 E5 2A 14 45 22 14 5B 3E 0A CD 40 40 E1 C9 00 00
D09,3E=ED 5B 14 5B 3E 0A CD 3D 40 3E 00 32 89 42 C3 2D 40
.. End of patch SCRIPIII/FIX.
```

```
.SCR17III/FIX
.LDOS Patch to SuperScripsit by Tom Price and Renato Reyes
.Permits operation under LDOS 5.1.3 Mod III
.This patch compliments of PowerSOFT
.
.PATCH SCR17/CTL USING SCR17III/FIX
D02,40=38
D02,46=D6 30 4F 06 00
D02,DA=37
D04,69=4C 44 4F 53 20 20
... End of patch SCR17III/FIX.
```

Model I

```
.For MODEL I SuperScripsit. Apply the patch to Model I SCRIPSIT/CMD
.
.SCRIPI/FIX
.LDOS patch to Model I SuperScripsit by Tom Price & RB Reyes
.Permits operation under LDOS 5.1.3 Mod I
.This patch compliments (f PowerSOFT
.
.PATCH SCRIPSIT/CMD USING SCRIPI/FIX
.
D00,08=3A 1F 44 32 20 5B CD 01 5B 00
D05,2E=C3 16 5B
D09,07=47 0E FF 21 9B AC E5 CD 96
```

```
D09,10=43 E1 C0 23 23 46 23 7E B7 3E FF 20 03 00 00 78
D09,20=32 22 7E AF
D09,28=C9 CB 8F 32 1F 44 E5 2A
D09,30=14 45 22 14 5B 3E 0A CD 13 44 E1 C9 00 00 ED 5B
D09,40=14 5B 3E 0A CD 10 44 3E 00 32 1F 44 C3 2D 40 00
...End of patch SCRIPI/FIX
```

If you are running a double-density Model I system under LDOS 5.1.3 you can also get directory display capability under Model I SuperScripsit by following these steps:

1) Copy the Model III SCR17/CTL to your Model I Super Scripsit disk. Also copy the Model III HELP/CTL module to your Model I system.

2) Patch the SCR17/CTL module which you just copied as follows:

```
.SCR17HY/FIX
.Patch to Model III SCR17/CTL module for use under Model I
.SuperScripsit running double-density LDOS 5.1.3/I.
.This patch compliments of PowerSOFT
D02,40=38
D02,46=D6 30 4F 06 00 CD 63 44
D02,DA=37
D04,69=4C 44 4F 53 20 20
...END OF PATCH.
```

The use of this module will also permit you to use the Mod III dictionary and proofreader in place of the Mod I's. Just copy over the Mod III PROOF/CTL to your model I SuperScripsit disk along with the rest of the dictionary files for Mod III.

No patching of PROOF/CTL needed. Be careful not to mix dictionaries as they use different encoding schemes.

JCL File for automatic applicaton:

```
.SCRLDOS/JCL - Automatic Applicator for SuperScripsit Patches
.for use with LDOS 5.1.3 - Compliments of PowerSOFT
.This file will modify Radio Shack's SuperScripsit to run
.under LDOS 5.1.3. The SuperScripsit files must have already
.been transferred onto an LDOS 5.1.3-readable disk.
.Has this been done?
//Keyin 1 if yes, 2 if no ==>
//2
Please do this now.
//Exit
//1
.Which system do you wish to implement:
.
.              (1) Model III
.              (2) Model I
.              (3) Hybrid Model I with directory
.                  display
.
//Keyin your choice (1, 2, or 3) ==>
//1
Patch Scripsit/Cmd using SCRIPIII/FIX
Patch SCR17/CTL using SCR17III/FIX
```

```
.Your SuperScripsit is now usable under LDOS 5.1.3-III
//exit
//2
Patch Scripsit/cmd using SCRIPI/FIX
.Your SuperScripsit is now usable under LDOS 5.1.3-I.
//exit
//3
.This requires that you copy the following MODEL III files
.onto your Model I SuperScripsit disk.
.           (1)  SCR17/CTL
.           (2)  HELP/CTL
.Has this been done?
//Keyin 1 for yes, 2 for No ==>
//2
.Please do this now and rerun this JCL.
//exit
//1
Patch Scripsit/cmd using SCRIPI/FIX
Patch SCR17/CTL using SCR17HY/FIX
.
.You now have a hybrid Model I SuperScripsit system with
.directory display implemented. You may also use the Model III
.Proofreader with this implementation. Copy the Model III
.Proof/Ctl module onto your Model I system and use the Model
.III Scripsit dictionary.
.
//exit
```

SuperScripsit(tm) is a registered trademark of the Tandy Corp.

ITEMS AT RANDOM

The following pieces of news and other information were received too late to be placed in the regular part of the newsletter, so they are here at the end.

First, available from SoftERware is an MX-80/100 printer driver for the new SuperScripsit program. It handles different functions for the Epson printers that have the Graftrax option installed. The price is $18.95, and the driver and information on it can be obtained from SoftERware at 16007 Miami Way, Pacific Palicades, CA 90272.

Bob Snapp has reduced the prices of his SNAPPWARE products for the LDOS system. These products, as well as his trial package, are available from Logical Systems or direct from Snapp at 3719 Mantell Ave., Cincinatti, OH 45236. The new prices are:

SNAPP-II	$39.00
SNAPP-III	$35.00
SNAPP-IV	$35.00
SNAPP-V	$29.00
SNAPP-VI	$35.00
SNAPP-VII	$19.00
TRIAL PACK	$10.00

An upper case version of LED will soon be available for those Model I owners who do not have a lower case modification in their CPU. It will come standard on the LED master disk.

The LDOS Quick Reference card listed in our price list under catalog # L-40-060 should be available by the end of October. This will be a typeset, 3 color, glossy card with 10 panels per side. It will detail the LDOS Library comands, filters, drivers, utilities, LBASIC statements and errors, and several other charts. The price is $5.95 which includes shipping.

The following is a series of patches by Richard Deglin for the Microsoft Macro-80 Editor Assembler package. For a complete explanation, refer to the comments in the patch files. The patches from the October Quarterly that are referrenced in this article can be found on the fix disk, as explained at the end of this section.

```
. M80TITLE/FIX
. Patch to Microsoft M80 3.43
.  Written by Richard N. Deglin 08/l0/82
. (1) Extracts system date and places it in M80 title
. (2) Prints M80 title as logon message
.  Requires previous patch M80/FIX, LDOS Quarterly, Oct 1981, page 17!
X'5216'=C3 C2 97
X'97C2'=E5 CD 5D 96 20 06 21 33 30 22 D2 97 21 25 98
X'97D1'=CD 70 44 2A 28 98 22 1C 6B 2A 2B 98 22 23 6B
X'97E0'=2A 25 98 7D D6 30 06 00 28 02 06 0A 7C D6 30 80 3D 47 87 80
X'97F4'=21 2D 98 16 00 5F 19 11 1F 6B 01 03 00 ED B0 3E 0D
X'9805'=32 25 6B 3E 20 32 1B 6B 21 0D 6B CD 67 44 3E 09
X'9815'=32 25 6B 32 1B 6B 21 DB 92 22 17 52 E1 C3 DB 92
X'982D'="JanFebMarAprMayJunJulAugSepOctNovDec"
. EOP

. patch MACRO80/CMD Version 3.34 - 02/24/82
. allow lowercase listing output (strings, titles, comments, etc)
X'914B'=00
. EOP

. patch MACROB0/CMD Version 3.43 - 06/20/82
. allow lowercase listing output (strings, titles, comments, etc)
X'965B'=00
. EOP

. patches to EDIT80/CMD l.0 - 02/24/82
. (1) change P<CR> command to scroll only 15 lines
X'69BF'=15
.(2) change all listing formfeeds to carriage returns
X'66FD'=0D
X'6C35'=0D
X'878D'=0D
. (3) enable lowercase commands
X'624C'=38 03 DE 20 77 F1 7E 23 30 15
. EOP

. patch CREF80/CMD Version 3.43 - 07/31/82
. allow lowercase listing output (strings, titles, comments, etc)
X'5C37'=00
. EOP
```

FIXES and the FIX DISK

Logical Systems has a disk available that contains the major patches that we and our users have assembled to date. This disk is available for 0 from LSI. A hardcopy only version is also available for 0 The current contents of the disk are as follows:

The date of this release is 10/01/82.

***** FIX/TXT - Contains instructions for the SCRIPT, LSCRIPT, and VC patches.

***** SCRIPT1/FIX - Makes Model 1 SCRIPTSIT, Version 1.0, work with Model I LDOS.

***** SCRIPT3/FIX - Makes Model 1 SCRIPSIT, Version l.0, work with Model III LDOS.

***** SCRIPT32/FIX - Makes Model III Scripsit Version 3.2 work with Model III LDOS.
Unlike the Script3 fix, no additional features such as a directory query or use of the spooler are supported.

***** LSCRIPT/FIX - Enhances Model 1 SCRIPSIT, Version l.0, for use on either Model I or III LDOS.

***** PENCIL/FIX - Lets ELECTRIC PENCIL, Version 1, work with Model I LDOS.

***** VC/FIX - Makes Model I Visicalc, Version 1.20Z, work on Model I or III LDOS.

***** RSCOBOL/FIX, RUNCOBOL/FIX, CEDIT/FIX - Makes Radio Shack COBOL work on either Model I or III LDOS.

***** RSBASIC/FIX, BEDIT/FIX - Makes Radio Shack BASIC Compiler work with Model I or III LDOS.

***** EDIT80/FIX, LINK80/FIX, CREF80/FIX, M80/FIX - Makes Model I Microsoft MACRO-80 Assembler run on either Model I or III LDOS.

***** BASCOM/FIX, BRUN/FIX, LINK80B/FIX - Makes Model I Microsoft BASIC compiler run on either Model I or III LDOS.

***** FORLIB/FIX, F80/FIX, LINK80/FIX, EDIT80/FIX - Makes Model I Microsoft FORTRAN run on either Model I or III LDOS.

***** DTPLAN/MRG - A series of fixes for the BASIC Desktop Planner program from Radio Shack.

***** MLS/MRG - Fixes the MLS program of Radio Shack's Business Mailing List program.

***** VC31/FIX - Fixes Model III Visicaic Version 3.1Z for use with Model III LDOS.

***** VC3l5/FIX and VC316/FIX - Fixes Model III Enhanced Visicalc Version 150Y0 and
160Y0 for use with Model I or III LDOS.

***** SCRIPI/FIX, SCRIPIII/FIX, SCR17/FIX, SCR17HY/FIX - Patches for SuperScripsit to run on the Model I and III with LDOS.

***** EDITIII - Fixes Model III FORTRAN editor EDIT/CMD file loading.

QuizMaster

QuizMaster is an educational/informational question and answer program and can also be used as a game. Its basic operation is to display a question and four possible answers. It scores the operator's response based upon the speed as well as correctness from one of three possible skill levels.

QuizMaster randomizes the order of the answers to prevent memorization. The question sequence is never the same. Extended play provides a "sudden death" feature for the skillful user.

QuizMaster comes with three subject files of 100 questions each, U. S. Information, General trivia as well as Fantasy and Science Fiction trivia. These files can be increased or edited, or the user's own specialty files can be created and utilized. Each file can hold up to 255 question/answer sets and the only limit to the number of files is the number of diskettes you possess.

QuizMaster is educational, interesting and addictive. QuizMaster runs under the LDOS operating system to utilize maximum efficiency. The QuizMaster system includes all the facilities necessary to establish and maintain a series of multiple choice questions on any subject whatsoever. The system is comprised of several machine language modules for fast and accurate access and response times.

Word Processor-Like Input Editor

For ease of entry an "input editor" allows full transparent cursor motion along with insert and delete modes, type over and fast cursor positioning. This feature is found in both the "Add" and "Edit" modes.

Five Support Programs Included

Five support programs are provided to create, extend, edit, print and maintain question/answer files. Also included is a program to reconstruct a file that has been damaged by disk I/O errors or faulty disk media. A packing module allows files that have been heavily edited to be compressed and use disk space more efficiently.

All features are easy to use and easy to operate. Everybody loves trivia and now you can control it. Other uses include

- *** classroom testing
- *** procedure quizzes
- *** product knowledge and
- *** group entertainment

QuizMaster can be ordered now for just $39.00

LOGICAL SYSTEMS INC.

11520 N. Port Washington Rd.
Mequon, WI. 53092
(414) 241-3066

The BASIC Answer

The BASIC Answer is a BASIC text processing utility. It is designed to allow the BASIC programmer to build code in a structured manner. "Source" code is written with a word processor or text editor which allows the user to exploit the powerful editing and movement features characteristic to those types of editors. Source code can even be created by your own BASIC interpreter. The BASIC Answer is then used to process these files into normal interpretive BASIC code.

Free yourself from line numbers

The BASIC Answer allows substitution of labels for line numbers! This means that your BASIC code now can read like a novel. Instead of the typically undescriptive "GOSUB 1000", a label such as "GOSUB @Search.Name" is used. Imagine yourself reading code filled with such descriptive branches and understanding it at a glance, even years later. This feature even allows totally relocatable BASIC routines without the renumbering problems.

A New Concept in Variable Usage

The BASIC Answer allows variable names to be as long as 14 characters and ALL 14 are significant. Imagine reading

"IF ACCNT.OVERDUE# > 0 THEN GOSUB @PRINT.DUN"
or
"IFAO#>0THENGOSUB52130"

Which would you rather read? It also introduces to BASIC the concept of Global and Local variables. This feature circumvents the tedious problem of variable tracking because a Local variable is only viable in its own subroutine!

End the Multiple Machine Hassle

The BASIC Answer introduces the concept of "Conditional Translation." This feature allows the programmer to place different "machine dependent" code simultaneously into the same Source Code. The BASIC Answer can be "switched" when processing to ignore the unwanted or include extra code! No more multiple master programs to confuse maintenance. All the masters could now be rolled into the same program. Modify the one master and you've modified them all. Process the same code with different switches set, and get two or more versions from the same source.

The BASIC Answer combines the self-documenting power of COBOL with the relative ease of BASIC together with the power of a word processor.

The BASIC Answer is available for just S69.00

******* The BASIC Answer - - - Source Code Sample *******

```
'Prime Number Detection Routine
'Detects primes from specified input ranges

=Primes%,Start%,End%,End$,Step%,LOOP1%
     DIM Primes%(25):GOTO @ORIGIN
@INIT.DEVISORS=Counter%,Mark%,LOOP1%
     ?">>> Initializing <<<":Counter%=3: Mark%=1: Primes%(0)=2: Primes%(1)=3
@Test.Prime
     Counter%= Counter% + 2
          FOR LOOP1% =2 TO INT(Counter%/2)
               IF Counter%/LOOP1% = INT(Counter%/LOOP1%) THEN @Test.Prime
          NEXT LOOP1%: Mark%=Mark%+1: Primes% (Mark%)= Counter%
       IF Mark%=24 THEN @END.IT ELSE @Test.Prime
@END.IT
     RETURN
@DISPLAY.100
     FOR LOOP1%=0 to 24: ?USING"##,###"; Primes%(LOOP1%);
          ?" * Prime *",
     NEXT LOOP1%: Start%=101
     RETURN
@ORIGIN
     CLS: ?TAB(25)"Prime Number Detection"
     ?: INPUT"Start at value"; Start%
     INPUT"End at Value (Enter < 32767)"; End$
       IF VAL(End$)> 32767  THEN End% = 32767 ELSE End%= VAL (End$)
       IF End% < Start% THEN Step%=-2 ELSE Step%= 2
       IF Start%/2= INT(Start%/2) Then Start%= Start% +1
     GOSUB @INIT.DEVISORS
       IF Start%<100 THEN GOSUB @DISPLAY.100
     FOR LOOP1%= Start% TO End% STEP Step%
          GOSUB @TEST.FOR.PRIME
     NEXT LOOP1%: END
@TEST.FOR.PRIME=HI.FACTOR%,LOOP2%,Prime%
     HI.FACTOR%=INT(LOOP1%/2)
          FOR LOOP2%= 24 TO 0 STEP-1
               IF LOOP1%/Primes%(LOOP2%)=INT (LOOP1%/Primes%(LOOP2%)) THEN Prime%=1:
               GOTO @Display
          NEXT LOOP2%
          FOR LOOP2%= Primes%(24) to HI.FACTOR%
               IF LOOP1%/LOOP2% = INT (LOOP1%/LOOP2%) THEN Prime%=1: GOTO @Display
          NEXT LOOP2%: Prime%=2
@Display
     ?USING"##,###";LOOP1%;
     ON Prime% GOTO @NO,@YES
@NO
     ?" Not Prime",: GOTO @LEAVE
@YES
     ?" * Prime *",
@LEAVE
     RETURN
```

******* The BASIC Answer - - - Object Code Sample *******

```
5 DIMPS%(25):GOTO22
7 ?">>> Initializing <<<":CO%=3:MA%=1:PS%(0)=2:PS%(1)=3
9 CO%=CO%+2
10 FORLO%=2TOINT(CO%/2)
11 IFCO%/LO%=INT(CO%/LO%)THEN9
12 NEXTLO%:MA%=MA%+1:PS%(MA%)=CO%
13 IFMA%=24THEN15ELSE9
15 RETURN
17 FORLQ%=0to24:?USING"##,###";PS%(LQ%);
18 ?" * Prime *",
19 NEXTLQ%:ST%=101
20 RETURN
22 CLS:?TAB(25)"Prime Number Detection"
23 ?:INPUT"Start at value";ST%
24 INPUT"End at Value (Enter < 32767)";EN$
25 IFVAL(EN$)>32767THENEN%=32767ELSEEN%=VAL(EN$)
26 IFEN%<ST%THENSU%=-2ELSESU%=2
27 IFST%/2=INT(ST%/2)ThenST%=ST%+1
28 GOSUB7
29 IFST%<100THENGOSUB17
30 FORLQ%=ST%TOEN%STEPSU%
31 GOSUB34
32 NEXTLQ%:END
34 HI%=INT(LQ%/2)
35 FORLP%=24TO0STEP-1
36 IFLQ%/PS%(LP%)=INT(LQ%/PS%(LP%))THENPR%=1:GOTO42
37 NEXTLP%
38 FORLP%=PS%(24)toHI%
39 IFLQ%/LP%=INT(LQ%/LP%)THENPR%=1:GOTO42
40 NEXTLP%:PR%=2
42 ?USING"##,###";LQ%;
43 ONPR%GOTO45,47
45 ?" Not Prime",:GOTO49
47 ?" * Prime *",
49 RETURN
```

LOGICAL SYSTEMS INC.

11520 N. Port Washington Rd.
Mequon, WI. 53092
(414) 241-3066