THE MISOSYS QUARTERLY

Look at what is in this issue:

Get your computer working for you

Volume VII.iii **$10** **Summer1993**

Let LB Data Manager solve your data storage problems

LB Version 2.3: Modern up-to-date features provide this newest release of our Flat File Data Manager with a greater degree of flexibility and an increased level of ease-of-use. LB still provides strong data base capabilities with absolutely no user programming!

NOW WITH COLOR SUPPORT FOR PC USERS

We've added many features asked for over the past few years by LB users; yet LB is still about the easiest, most flexible data manager you can use for managing your data. Absolutely no programming is needed to create a database with up to sixty-four fields; construct input screens for adding, viewing, and editing data; and create your own customized report. Quickly you define your data fields in response to LB's prompts, and then draw your data input screen using simple keystrokes - or have LB automatically create your input screen. In no time at all, you're entering data. Customize your printed reports with user-definable print screen definitions. Or use LB's define print autogen module to automatically create Table or Form reports, or over a dozen different address label configurations including a Rolodex™ card and a 3" by 5" index card. LB is just what you need in a data manager! We even have many database templates available for download on our Compuserve forum!

Data capacity per database:

LB supports up to 65,534 records per data base; 1,024 characters (64 fields) per record; and up to 254 characters per field.

Field types supported:

LB allows ten field types for flexibility: *alphabetic* {A-Z, a-z}, *calculated* {operations on "numeric" fields using +, -, *, /, with 2-level of parentheses}, *date last modified* {YYYY/MM/DD automatically maintained}, *dollar* {±ddddddd.dd}, *floating point* {±ddddddd.ddddddd, *literal* {any ASCII character}, *numeric* {0-9, -, .}, *right-justified numeric* , *upper case alphabetic* {A-Z, automatic conversion of a-z}, and *upper case literal* { literal with automatic conversion of a-z}. All field types utilize input editing verification so invalid data cannot be added to a record. Field name strings can be up to 19 characters long.

Data entry and editing:

LB allows you to design up to ten different add/update view screens to provide extreme flexibility for selectively viewing your database fields. You can customize the appearance of any view screen by a simple drawing process, or use LB's built-in *autogen* capability. View screen definition even provides an intelligent line-drawing mode so you can create lines and boxes to enhance the appearance of your screen image. If your computer supports a color video adaptor, each view screen can be configured to a distinct foreground/background color arrangement to increase the distinction of it's data viewing.

Using a database password provides the capability of selectively protecting fields from being displayed or printed without entry of the correct database password, or they can be protected from being altered. This is quite useful in a work-group environment. Fields may be selectively established to require a data entry before a record being added or edited is saved. You can enable a special index file to keep track of records being added. This can be subsequently used, for example, for a special mailing to newly added *customers*. Flexible editing includes global search and replace with wild-card character match and source string substitution. Search and replace can be performed on all records, or on records referenced in an unsorted or sorted index file.

Record selection and sorting:

You can maintain up to ten different index files to keep your data organized per your multiple specifications. Records may be selected for reference in an index file by search criteria using six different field comparisons: EQ, NE, GT, GE, LT, and LE. You can select on up to eight different fields with AND and OR connectives. Index files can be left unsorted, or you can sort in ascending or descending order. By associating a sorted index file, any record can be found within seconds - even in a very large database.

LB even includes a special *Dup* command which uses the sophisticated Ratcliff/Obershalp pattern recognition algorithm for automatically finding duplicate or near-duplicate records! Duplicates can then be either manually deleted or automatically purged using the provided LBMANAGE utility program.

Automatic operation:

For automating your processing needs, LB can be run in an *automatic* mode, without operator intervention. Frequently used procedures can be saved by LB's built-in macro recorder for future use. Entire job streams may be produced, so that LB operations may be intermixed with literally any DOS function that can be *batch* processed. These named procedures are easily invoked via a pop-up list-box.

Maintenance utilities:

To make it easy for you to grow your database as your data needs grow, we provide three utility programs for managing your database. LBREDEF allows you to construct a new database with an altered data structure and populate it with data from your existing database. This facility is great for adding new fields, or deleting fields no longer needed. Or you can use LBREDEF to redefine the field type of an existing field and convert the existing data. Another utility, LBMANAGE, allows you to duplicate your database structure, copy or move records from one to another, or automatically purge un-needed records.

A third utility, LBCONV, converts to LB from pfsFILE4, Profile4, DIF, dBASE II, dBASE III, and fixed record. It also converts to DIF, dBASE, and tab or comma delimited files to enable easy porting of LB data to other systems.

Report generation:

Report generation incorporates a great degree of flexibility. Your report presentation can be totally customized through print definition formats which you define on the screen as easily as you define the add/update view screens. You can truncate field data, strip trailing spaces, or tab to a column. You control exactly where you want each field to appear on your report. LB provides for a report header complete with database statistics: database name, date, time, and page numbers. A report footer provides subtotaling, totaling, and averaging for calculated, dollar, floating point, and numeric fields; print number of records printed per page and per report.

Many report formats can be automatically created by LB's define print autogen module. You specify your printer type and character size from a pop-up list-box. Select one of four canned report formats: narrow or wide carriage Table reports; a Form report; or an address mailing label using one of six different sizes of labels including labels printed two, three, and four across. Label formats also include formatting for a Rolodex™ card and a 3" by 5" index card. Label formats automatically select the needed fields from your database definition.

For printing, associate any of the ten index files and you control exactly what records get printed; even a subset of indexed records can be selected for printing to give you a means of recovering from that printer jam halfway through your 30-page printout. You can even force a new page when the key field of an index file changes value. Up to ten different printout definition formats can be maintained for each database. Reports may be sent easily to a printer, the console display screen, or to a disk file - useful for subsequent printing or downstream data export to other programs. Report formatting allows for multiple across mailing labels, multiple copies of the same record, or even *form* printing one record per page for sales books. You can easily generate mail/merge files of address or other data for your word processor. Or you can use LB's built-in form letter capability.

Help is on the way:

The main menu even provides a shell to DOS so you can temporarily exit LB to perform other DOS commands. LB provides extensive on-line help available from almost every sub-command. A 200-page User Manual documents every facet of LB's operation.

Competitive Trade-up policy:

Send in an original Table of Contents page from any existing database program and get LB Version 2 for half price. That's only $49.50 + S&H!

Ordering Instructions

Specify MS-DOS (and media size) or TRS-80/4 version. LB is priced at $99 + $5 S&H US ($6 Canada; $7 Europe; $9 Asia, Pacific Rim, and Australia).

MISOSYS, Inc.
PO Box 239
Sterling, VA 20167-0239

703-450-4181 or orders to 800-MISOSYS

The MISOSYS Quarterly is a publication of MISOSYS, Inc., PO Box 239, Sterling, VA 20167-0239, 703-450-4181.

THE MISOSYS QUARTERLY

Subscriptions are no longer being accepted for *The MISOSYS Quarterly.*

TMQ Toolbox

The MISOSYS Quarterly is published using the following facilities:

The hardware used to produce the "camera ready" copy consists of an AST Premium/386 computer (20 MHz) with 9 Megabytes of RAM, a Seagate ST4096 80M HD, ST251 40M, Expanz! card; a CMS DJ10 tape backup, a NEC Multisync II monitor driven by a Video Seven VGA card, an AST TurboScan scanner (Microtek MS300), and a NEC LC-890 PostScript laser printer.

Text is developed, edited, spell-checked, and draft formatted using Microsoft WINWORD Version 1.1; Submissions on paper and letters are scanned and converted to text using ReadRight optical character recognition software by OCR Systems. Final page composition is developed using PageMaker 4.0 by Aldus.

Table of Contents

List of Advertisors

List of Patches/Updates in this Issue

TMQ Index

In this column of the last issue, I announced the availability of an index to all past issues of *The MISOSYS Quarterly*. In the intervening months, I have received orders for two printed copies and no disk copies. Because of the dismal demand, the index is discontinued.

Upcoming at MISOSYS

As this is being written, the conversion of LB86 MS-DOS version database to a DeskMate environment is approximately 50% complete. It appears that LBDM² will be either a one or two program package (not counting utilities). I have completed the database definition module, the screen definition module, the entire pull-down menu-driven front end, the screen presentation, the ability to view and edit path file definitions, and the ability to view field data. Essentially left to implementation is the sort and select module, the print screen definition, and the report writer.

LBDM² uses the identical database structure as LB and LB86; however, it was apparent that to provide graphical features in the view screen, it was necessary to design a completely new view screen file structure. LBDM²'s screen is object-oriented; it currently supports five different object types: lines, beveled rectangles, text strings, data fields, and DeskMate DRAW figures. The screen designer has total control over the color, width, and type of each individual line. The designer also has complete control over the color, width, bevel, fill, and pattern for rectangular objects (beveled rectangles collapse to a rectangle with a bevel of zero). Each text string can have its own foreground and background color as well as character attributes (bold, underline, inverse, and grayed. All data fields currently supported by LB are supported in LBDM². Data field viewing, however, uses DeskMate *edit fields*. Because of the capability of built-in scrolling, a maximum horizontal window of 64 characters in width is provided with the field contents scrollable within the view window. Fields can also be bordered with a flat box or raised box (or no box) at the user option. Any number of DRAW figures can also be placed on the view screen.

Rather than have specific upper limits on the quantity of objects which can be placed on a screen, LBDM²'s screen files are variable length; a data structure defines the number of each object type. During screen definition, the designer can select and move an object about the screen for placement revision (or deletion). Currently select, move, and relocate is the means by which objects are relocated, however, object drag and drop will be implemented. Selected objects can also be edited as to their individual characteristics: color, width, pattern, type, etc.

With but one exception, any time a field reference is needed, it is selected from a pop-up dialog box with the list of fields referenced in a list box. The one exception is the field reference within a calculation string.

The screen designer also includes an automatic design option which is similar to the autogen feature in LB86. The option has been enhanced to configure a view screen with either one or two columns of data fields - automatically limiting each data field's view window to fit within the allocated region.

The Blurb

by Roy Soltoff

Depending on the shift of my time to outside employment, I may not reach my target of a summer release of LBDM²; however, it should make it out the door this year. There will be some features of LB86 which won't make it in to LBDM² - such as "Run Automatically". But to compensate, LBDM² already has features that are not in LB86. There's always a trade-off.

Points to Ponder

According to an article in *Electronic Engineering Times*, trillions of bits per square inch in data storage may soon reach fruition based on research at Kyoto University using photochemical hole-burning techniques (PHB). Kazuyuki Hirao, the researcher, worked at room temperature with boric acid glass doped with samarium ions. A variable wavelength laser is used to excite electrons in the medium which changes their absorption characteristics resulting in a "hole". Bit detection is performed when light passes through without being absorbed.

The boric acid glass is widely used in the manufacture of heat-resistant kitchenware; thus, production techniques are not difficult. However, further technological advancements must be made in the development of lasers whose wavelengths could be stepped in 0.01nm, as well as an increase in the read/write speed (currently about one second).

According to Dave Webb, writing in *Electronic Buyers News* in response to Intel's

recent $2 Billion in revenue per quarter, "In the old days, IBM used to make more in profits than anyone else did in revenue. It's entirely possible Intel is coming into that mode." Now I don't know about you, but a profit of over $500 million on revenue of $2 billion for three months means a huge number of processor sales. Seventy-five percent of their revenue comes from CPUs. Too bad IBM dumped most of their Intel stock a handful of years ago. Why in just one year, Intel's revenue could equal the net worth of Bill Gates!

I found it interesting to read that Japanese companies supply 60% of the world's color CRT's, and 100% of the 15-, 17-, 20-, and 21-inch color CRT's. That's why a recently announced price increase of 6% has sent shock waves around the world. Big waves hit Taiwan, which now manufactures 47% of the world's color and monochrome monitors. So look for monitors to edge up in price.

Did you know that the electronics industry uses about 7% of the lead in the United States? That stems from use in solder, wire, cable, dry-cell batteries, and radiation shielding in CRT's. The electronics industry trade associations are therefore fighting a 45-cent per pound lead tax being proposed by Representative Cardin (D-Md). The tax is designed to remove lead-based paint from child-care centers and residential structures.

I mentioned in a previous Blurb that computers may one day be at the bottom of the heap in the world's use of RAM chips. Well, it appears that computers may also lose their dominance in the use of high-powered microprocessors. CPUs have already been widely used in all sorts of electronic equipment in embedded applications. CPUs will now be migrating to the TV set and cable convertor boxes.

Intel announced they are joining with General Instrument Corporation and Microsoft (in what business are they not placing their tentacles?) to develop smart cable converter boxes using 386 chips for interactive TV. Most likely, a windows-type on-screen interface will be provided to enable TV viewers to "easily navigate among potentially hundreds of cable channels and services".

How much music do you get on an audio CD? In most cases, it's approximately 75 minutes maximum. But look out, a Japanese petrochemical company has developed a new synthetic resin which should permit ten times as much material to be recorded on a CD-sized disk. The new thermo-plastic X, called TPX for short, has sixteen times less viscosity than the polycarbonate structure currently used in CDs. That should permit the imprinting of data at a much higher track density.

I remember when a perm meant a trip to the hair salon. Not so for Sony! Their new Pre-embossed Rigid Magnetic technology (PERM) is a process whereby servo information is written into grooves cut into the surface of a disk. The facility allows increased track density providing a prototype 2.5" platter with 5,000 tracks per inch - or a capacity of 200 megabytes. That's just a little greater than the width of the JKL keys. They expect to eventually increase the track density to 15,000 tpi.

Remember pricing on the TRS-80 Model I? The base 16K Level II was $999. Memory additions of 16K were in the range of $250. The expansion interface was $299. Radio Shack floppy drives were $499. You even had to pay $16.95 for a three pack of single-density diskettes. I remember a few of us in the computer club grouped together to buy diskettes directly from Verbatim in boxes of 100 - even then they were hard to get. I recollect feeling like I suspect a drug dealer would feel making these clandestine calls - "Hey, got any in yet?" List price on a Line Printer III was $1960; I got a used one closed out with a bad printhead for $10. Remember a 64K Model II for $3899?

Well DEC's new 64-bit Alpha PC with a 150 megahertz processor, 32 megabytes of RAM, 600 megabyte CD-ROM, Ethernet LAN port, a 3.5" 2.88 megabyte floppy, a high-resolution SuperVGA monitor, and a 426 megabyte SCSI hard disk drive was unveiled at Windows World in late May for a price of $6995. The Alpha processor is a 300 million instructions per second chip. Douglass Hamilton, president of Hamilton Laboratories Inc., developer of a UNIX development tool environment, "This machine is dynamite. It's faster than any machine I've ever worked on."

So what do you have to say about the sale of Tandy's computer manufacturing business to AST Research?

Trade-in Policy

With the closeout of most TRS-80 products, our trade-in policy exists solely for our LB database and remaining MSDOS-related products. The policy, where applicable, is to just send in an original *Table of Contents* page from an equivalent non-MISOSYS software product with the trade-in fee which is 50% of the price of our product. So for LB 2.3, trade in any other database product and you can purchase LB or LB-86 for $49.50 plus S&H. How's that for a deal? It doesn't matter for what system or operating environment your trade-in was designed for. This offer does not extend to products re-sold by MISOSYS or products on sale.

In this issue...

Continuing with the C venue, though not in tutorial form, I bring you a useful ringmove algorithm for optimizing the reordering of index-sorted data. There's also an article which sheds light on a obscure problem when using DMA on a PC. A handful of contributed entries make up the remainder of this next to last issue.

TMQ Schedule

Folks, there's but one more issue of *The MISOSYS Quarterly* left to go. Anyone who has a "**93/08**" on their mailing label will be sent that issue in approximately three months. TMQ will cease publication with issue VII.iv. I currently have some copies of Volume VI which can be had for $4 per issue - plus S&H.

MISOSYS Forum

I sponsor a forum on CompuServe. You can reach some "experts" on TRS-80 and MS-DOS subjects by dialing in, then GO PCS49, or GO LDOS. This is probably the oldest forum still-surviving from the *MicroNet* days. If you want to see it continue, how about popping on for a chat or a question.

The forum contains many programs to download, as well as lively discussions which thread through the message system. You can direct a message to me at 70140,310. Post a message in private if you don't want it "broadcast".

DISK NOTES 7.3

Each issue of *The MISOSYS Quarterly* contains program listings, patch listings, and other references to files we have placed onto a disk. Where feasible, the text accompanying an article is also on DISK NOTES. DISK NOTES 7.3 corresponds to this issue of TMQ. The disk is formatted usually for TRS-80 LDOS/LS-DOS users at 40D1 (that's 40 tracks, double density, one sided). MS-DOS users can request a 5.25" 360K disk. If you want to obtain the fixes and the listings, you may conveniently purchase a copy of DISK NOTES priced at $10 **Plus S&H. The S&H charges are $2 for US, Canada, and Mexico, $3 elsewhere.**

LB Templates

Please note the availability of the following LB database templates:

LB Template Disk 1

DRA	Dragon magazine article index
GAMEINV	Role playing game inventory
LEAP	Parent group address roster
PROP	Valuable property record
PTA	PTA roster
STAMPS	Stamp collection
STREK	Star Trek collection inventory
VID	Video Tape and Laser disk library

LB Template Disk 2

AUD	Catalog of audio disk/ tape collection
COMPUTER	Catalog of owned computer equipment
CREATURE	Catalog of adventure game creatures
LIB	Library card catalog
MAILFILE	Address mail list / LB database example
MISOSYS	Customer information database

To use any template, simply copy the files to your data drive, create a path file using LB menu option 14, then add your data. To create a template for others, simply use LBMANAGE to duplicate your database, then copy the new set of files to another disk. Submit your templates to MISOSYS for publication. They are available at $10 per disk + $3S&H, or free for download from our CompuServe forum. An MS-DOS 360K disk will hold a pair of template disks.

DOS Manuals

I will continue to publish both the "LDOS™ & LS-DOS™ Reference Manual" which covers LDOS 5.3.1 (Model I and III) and Model 4 LS-DOS 6.3.1, and the "LDOS™ & LS-DOS™ BASIC Reference Manual", which covers the interpreter BASIC which is bundled with LDOS 5.3.1 (even the ROM BASIC portion), the interpreter BASIC which is bundled with LS-DOS 6.3.1, and both Model I/III-mode and Model 4-mode EnhComp compiler BASIC. The DOS disks will continue to be made available. There are no more "Upgrade Kits".

MS-DOS Products

I still have some Tadiran TL-5296 6V lithium batteries usable in most AT-class machines. Don't wait for your battery to fail and lose your configuration data. A spare's shelf life will probably out last your machine.

LB86 continues to be available as a useful data base manager.

SCSI Driver

MISOSYS has a SCSI driver, H-HD-SWS, available for use with our H-HD-MHA Model III/4 host adaptor. The SWS driver is for directly supporting a Seagate SCSI drive or exact equivalent; it can handle a drive up to eight heads and 1226 cylinders (approximately 80 megabytes). Seagate drives which are in this capacity range include the 48MB ST157N, the 60MB ST177N, and the 84MB ST1096N. These drives are out of production, however, refurbished drives should be available at reasonable prices. Drivers for both Model III and Model 4 modes are included. As of this writing, three folks have purchased the driver. With that kind of sales volume, I might just retire to Tahiti!

FAX Number

If you want to reach us by fax, try 703-450-4213.

Closeouts

The big debate continues as to what to do with these old TRS-80 products. My closeout date of June 15th 1993 has come and gone. I am out of pre-printed documentation for a number of products: AFM, DoubleDuty, DSMBLR/PRO-DUCE, all three GO products (Maintenance, System Enhancement, Utility), HDPACK, Host/Term, MC/PRO-MC, MRAS/PRO-MRAS, PowerMail, RSHARD, Cornsoft Group Game Pack. These are all now discontinued. Other than the DOS versions, DOS manuals, and LB Database Manager, all other TRS-80 software products are discontinued. If I am able to find the time to rework documentation into a disk-file format, I will migrate selected products to shareware status. Now then, if folks want that process sped up, how about some volunteers who will be able to take documentation files along with program disks and build up archived sets of disks available for shareware duplication. You will need the capability of accessing documentation files written using SCRIPSIT, ALLWRITE, SuperScripsit, and Microsoft WORD.

Used Diskette Clearance

I have many cartons of used floppy diskettes - both 5.25" and 8". Most disks were used for archiving files and have been used very infrequently. All have labels affixed which are difficult to remove. Just to clean house, I will sell these disks dirt cheap. All disks will be bulk erased; sleeves are generally not available. Prices are as follows (shipping charges are additional)::

8" DS-DD	$0.40/each
8" SS-DD	$0.25/each
5.25" SS-DD	$0.15/each
5.25" SS-SD	$0.10/each

Hardware Clearance

Over the years, MISOSYS has accumulated TRS-80 hardware and related equipment in excess of current needs. The following items are now classified as surplus and are available for sale to the first takers (shipping charges are additional):

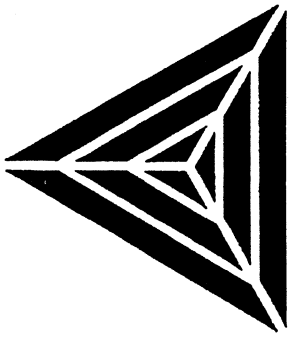
- Tandy color 2000 e/w stand $150
- Tandy 1000 & mono monitor $100
- TRS-80 Model 4P (gate array) $100
- TRS-80 Model 4P (gate array) $100
- TRS-80 Model 4D e/w XLR8er and MicroLabs Graphics board $150
- TRS-80 Model III (working) $25
- TRS-80 Model III (no video) $15
- Tandy DT-1 Data Terminal $15
- Amdek Video-300 Monitor $25
- BMC Monitor (for MAX or M1) $15
- 15 Meg Primary Radio Shack HD $150
- 12 Meg Secondary (bad drive) $25
- DMP-500 line printer $75
- Line Printer III (no printhead) $10
- Radio Shack Modem II $10
- Well-used bare floppy drives of various brands $5

Used Software

The following items of used software packages are available for purchase. These are items accepted as trade-ins or otherwise accumulated. **If you have an interest, I will sell all of these packages - approximately two cartons - for $50 + freight.** All items must go in the next three months or they hit the landfill.

- Radio Shack C, 26-2230 $15
- pfsFILE, Model 4 26-1518 $15
- pfsFILE, Model 3 26-1515 $15
- Series I EDTASM I/III 26-2013 $10
- Radio Shack ALDS 26-2012 $15
- Profile 3+ 26-1592 $15
- Compiler BASIC (III) 26-2204 $15
- ZEN EDTASM $5
- Quikpro+ $5
- ZBASIC 2.21 $5
- Level I BASIC Instruction Course $5
- Sargon II (cassette) $5
- Interlude (Model I) $5
- Gambiet 80 (mod-I tape) $5
- Macro-Mon (Model III disk) $10
- Personal Finance 26-1602 $10
- Blackjack/Backgammon (mint) $5
- Microchess 1.5 26-1901 $5

Letters to MISOSYS



Spurious Keystrokes

Fm Will Ramsey, Jr.: Dear Roy: I haven't written to you in a long, long time and I didn't want you to feel neglected or that I had forgotten about you!

I don't have any real problem or anything, so obviously this is a matter of extremely low priority, but this question has bothered me for years, so I thought I would finally go directly to the horse's mouth to get the answer (if there is one).

Since the very early days when we got our first Model 4 we have had this problem. We have had it with versions 6.0, 6.1, 6.2, and 6.3 of DOS. The problem is that somewhere in the keyboard software (at least I have always assumed it to be software, since we have the problem on all our computers) we get extra letters when certain keystroke sequences are typed.

Some of the most common are illustrated below:

We type:	The screen shows:
failure	faidlure
claim	claiem
plaintiff	plaifntiff
remain	remaifn
would	womuld

There are other examples, but these seem to be the most common. There seems to be a pattern in words with "ai" and "lai" in them, but that doesn't expalin "would" coming out "womuld". Also, generally the extra letter is an "f" but not always, as even these examples illustrate.

Several years ago I wrote 80 Micro and asked if anyone knew what caused this, but they never bothered to reply or print my inquiry in one of their open forum departments.

I have always felt that it might have something to do with our typing speeds. I type well over 100 WPM and our typists are in the 120 WPM range, and we use the computers almost exclusively for word processing. However, it seems that I have had this happen even when I was typing relatively slowly. It does happen in DOS as well as when we are using the word processing program (Scripsit Pro).

I don't know if you have ever run into this, but I would be interested to know your thoughts on it. Again, this is absolutely not critical, but some day when you're sitting around with nothing to do <<yeah, right!>> you might put on your thinking cap and think about this one for a while!

I hope all is well with you and yours, and I'm looking forward to the next TMQ. Best personal regards,

Fm MISOSYS, Inc: Dear Will: This is in response to your letter concerning extraneous characters. Your problem - and that of your typists - is that you all are typing too fast for the proper operation of the keyboard. Let me explain. When you press down one key of a keyboard, a circuit closure is sensed by scanning each row in turn then reading the result of the column. The data value then read can be decoded to a particular key given the row that resulted in a non-zero value. Okay, so what happens when you press down more than one key simultaneously. Depending on the way the keyboard is manufactured, you get a data value unexpectedly different from what you believe you should get.

Try this on your Model 4: Simultaneously - using three fingers - depress the keys [A], [I], and [L]. Do it repeatedly and you will find that some times, the letter "D" pops up. This is exactly the phenomenon that you are getting "faidlure" for "failure". Try pressing the triad, [A] [I] [N] and you will get an extraneous "F" at times; this is where you get "plaifntiff" for "plaintiff". Lastly, press the triad, [W] [O] [U] and you will pick up an extraneous "M".

All computer keyboards operate via a matrix: there are rows and columns where

each juncture represents a key. Look at page 13 of the Winter 92/93 issue of *The MISOSYS Quarterly* for an example of a keyboard matrix. Follow the keycaps of the domestic keyboard and you will see that the extraneous key which is generated falls at the fourth corner of a rectangle formed from the triad of keys identified above. What is happening is that the three keys are electrically depressed simultaneously resulting in a closure being sensed in the fourth corner. You can generate many other spurious key codes: press [S] [T] [L] and you get a "K" - not too common since that triad is unusual for a sub-string of a word. On the other hand, type the word [Q] [U] [I] [C] [K] too fast and the "ICK" is liable to generate a spurious "A".

I rarely ever get this kind of a problem because I never learned to touch type. Your only solution is to either slow down, or find a top-quality keyboard with much faster sensing circuitry and keys with less electrical contact bounce. Incidentally, this topic has been discussed to death years ago as the problem was infamous with the Model I keyboard.

Public Domain smallC

Fm MISOSYS, Inc: Frank Slinkman informed me a few months ago that David Goben released a small C compiler (I don't get CN80 so I have no way of keeping up with what Stan Slater has been publishing unless someone tells me - although I understand there has been a lot of spleen venting lately). I thought that odd as a version of Ron Cain's original smallC C compiler was available years ago in a public domain TRS-80 version. I told Frank that I would try to dig up the smallC version I remembered seeing in one of the many boxes of disks in my office. Well, on April 14, 1993 I found the copy of the Model 4 version of smallC and sent it along to Frank with the following cover letter for his possible disemmination of the disk to those having an interest.

Well, Frank, I located the smallC disk I was telling you about. I really had to dig back into the archives. It was a version of Ron Cain's (the original smallC) compiler adapted for TRSDOS 6.2. As you can tell by the file dates, it was available in December 1984 - almost ten years ago. Much documentation has existed in the past on using the compiler; most appeared in *Dr Dobbs Journal*, and there also were a few books available. I would suspect that anyone with a little research could dig up the reference material.

Have fun with this. The source code is there for the library and the compiler. So much for Goben's small C ten years after the fact. Where has he been?

Cheap Shots

Fm Richard James: Dear Roy, I'm enclosing the most recent copy of *The Misosys Quarterly* which was sent to my home address. I have already received one copy at my business address, and I don't need a second copy. I would appreciate your updating your records so that I receive just one copy.

I also want to let you know that the software you have distributed to the TRS-80 community (particularly LDOS) is of superb quality and is the equal of any software I have used on the TRS-80. I am dismayed at David Goben's cheap shots about your software. I have purchased two of his products and neither has been reliable. His PACK product will not work properly on any BASIC program which contains two consecutive null characters, although the DOSPLUS CRUNCH utility works fine. In addition, David's FBACKUP utility will not work reliably if called from within a BASIC program because it evidently doesn't respect the memory addresses used by BASIC. I reported the PACK problem in some detail to David, and his response was a sarcastic refusal to fix the problem.

These reliability problems are significant for me because I am providing software and support for about twenty TRS-80 users in my company. I have never experienced any reliability problems or a sarcastic attitude in my dealings with you, and I want to let you know that I appreciate what you have done.

LB and paths

Fm William E. Holmes: Dear Roy: Thank you for conferring with me on the phone yesterday concerning a small problem with LB Data Manager. After considering the matter, I have decided to leave my fields like they are, because having some of the last names out of alphabetical sequence with the first 5 digits of the ZIP is really of little significance. I mainly need them in ZIP order for mailing. If I need an alphabetical listing by club, I can easily get that by using CLUB as the primary select field.

We discussed briefly the ability to use a Temporary drive to hold my basic data file in order to speed up searching operations. It seems to me that the instructions under Menu #14 -Verify/Modify Path Settings are rather brief and don't seem to cover my situation.

First, let me describe my system. I am using a TRS-80 Model 4-D (two double-sided drives, 360K each), 64K RAM, with Anitek's Megamemory (2 MEG) installed. I have prepared a minimum system disk on which I have installed LB and MEGADRV

to initialize Anitek's megadrives. Upon startup I place this disk in the original Drive :0 (the lower one) and issue the command MEGADRV 0,10/A, which establishes a Megadrive :0 of 320K and copies all the files from the original drive :0 to it. The installed lower drive then becomes :1 and the upper one :2. I can establish Megadrives :3 and :4 at up to 352K each.

I have been using this configured sytem for the data base named LION, which presently has nearly 1,700 records, with drive :2 (the upper floppy) as the data file, drive :1 (lower floppy) for the index and Megadrive :3 specified as the working drive during sorting. Searching for a record on drive :2 seems to take quite a long time. I would like to use one of the Megadrives, but I want to be assured of having my data saved on a floppy disk, in case of power failure or other catastrophe. I can't decipher how to do this from the two pages on this subject in the manual. Please help me.

On another subject, which we also discussed briefly, I am considering procuring an IBM compatible computer, as there doesn't seem to be too much future for my TRS-8O's (I have two them - this 4-D and another one converted to two-sided drives, this one being used primarily for Amateur Radio Packet operations). I understand that I would have to have another Program disk, but I should be able to Hypercross my data files. Please give me a quote on the IBM compatible disk.

Thanks for helping this 74-year old fellow conquer LB to the extent of keeping files for making mailing labels. Previously, I have used ALLWRITE, which is extinct. I had to use five different files for my 1,700 records and had no sorting capability. I think I am going to like LB as I continue to learn some of its "tricks".

Fm MISOSYS, Inc: Dear William: This is in response to your letter concerning the use of a RAM drive for LB. What I was trying to say via telephone, was that the LB sort operation could be sped up considerably by establishing the temporary drive as a RAM drive. True, if you have the RAM drive capacity, you could speed up the entire operation of LB's data access by copying your data file (name/DEF and name/LB) to a RAM drive; however, if you are going to do any data updating, I really would not recommend that.

The sort process in LB requires the use of temporary storage if the amount of available memory (in the normal DOS address space) is insufficient to hold the sort strings and index pointers. It is the placement of the temporary work file(s) on a RAM drive which then speeds up the sorting. I believe that in your configuration, drive :3 is a RAM drive; you have properly established the PATH specifications for the Temp Path as ":3".

Now then, if you want to access your data from the RAM drive (without updating it), simply copy the two files noted above to your RAM drive prior to invoking LB; invoke LB and change the Data Path specification for the database to the RAM drive prior to selecting the database name; then select the database with menu option 1.

Lastly, if you wish to convert over to the MS-DOS version of LB (LB86), you can do so for half price ($49.50 + $5S&H).

Laser Printers for TRS-80s

Fm Martin J. Rapoport: I am constantly reading about how the Radio Shack Community needs to share information and ideas in order for us to maintain our Model IIIs and 4s. And yet, when I wrote a letter concerning my use of a laser printer with Model 4 Superscripsit, it was never printed. I am not looking for a pat on the back, but I have gotten many ideas and changes through the Radio Shack after-market network in general, and you in particular, and I wanted to give something back.

I am running my accounting office with two Model 4Ps, using Seagate 40 Meg drives (thanks to you), harddrive boot capability (thanks to TRSTimes), softboot to Model III mode (thanks to both), 360 5.25" and 720 3.5" floppies set up for 8 physical/logical drives. I use two MSDOS machines for all tax work, but I will not give up my Model 4Ps and Model 4.

Every article I ever read said that we cannot run Superscripsit with a laser printer because there are no drivers, and Superseripsit resets the printer if you preset it. This letter, and the letterhead, is being written on a Model 4, using Superscripsit and a laser printer.

I would think that there are a lot of users who would be interested in knowing how. If you wish, I can write an article detailing the key points. It is actually very simple for the knowledgable user. If not, then I will at least know that I tried. Incidentally, my own setup for the Epson LQ1050 works better then the Epson drivers I bought. I could not make them double underline, but my driver does. I have both the laser and Epson in the same driver so that I can actually run either one if I omit printer codes.

Fm MISOSYS, Inc: Sorry, Martin, but do to space limitations (and trimming a section to a page boundary) resulted in your last letter getting deferred. There is space in this issue to print it.

Now I have printed articles on using laser printers with the TRS-80. According to my copy of the TMQ index, I see *300 Dots on the TRS-80* in issue V.i, and *300 DOTS: An Update*, in issue V.iv. Also, there has been a few letters relating to Gary Shanafelt's and Dr Lee C. Rice's laserjet and deskjet utilities for Allwrite. It was also noted that Goben has a laser printer driver for Supercripsit and Scripsit Pro. But if you have something further to add, submit it and I will see if there is room for it. There's one more issue to go for *The MISOSYS Quarterly*.

SuperScripsit & Lasers

Fm Martin J. Rapoport, P.O. Box 315 Trexlertown, PA 18087 [215-398-1401]: It has been my understanding that the Model Superscripsit will not work with laser printers. The ones that might cannot double -underline. This letter is being printed using Superscripsit. The printer is a KYOCERA F-800A usin a Qume II emulation and legal character set.

I guess we are still finding ways to keep our Model 4's up-to-date with technology. I got lucky on this one, but it just extended the report preparation life of the Model 4 by about five more years.

A patch has to be made to the original DW2/CTL to change the double-underline character. The patch we made was:

Patch DW2/CTL (X'BC74 =1EBC)

We had previously made a patch to the same address to run a Star Powertype daisy wheel using a legal wheel. That patch was:

Patch DW2/CTL (X'BC74 =1E7C)

We have also made a patch that allows us to use the Epson LQ-1050 double-underscore in conjunction with printer codes to change the double underline sign from "=" to "(" and ")". I will make those available upon request free of charge.

You will also note that it will let you change fonts on the fly. I was not sure how well the printer would work with the Model 4, but I have done a set of financial reports and several letters quite easily.

The hard part is setting up the Superscripsit as I mentioned before. I wanted to confirm my findings with you and pass on the good news. TMQ and, of course, yourself, have been worlds of help to me and I figure I owe something back.

Anyone interested in the above can contact me and I can point them in the right direction. From a professional standpoint, I needed the ability to double underline the financial reports that I issue. It seems I have found a way to be as technologically proficient as the MSDOS crowd!

External Jumbo

Fm MISOSYS, Inc: We had a query concerning the usefulness of the external drive kit for the Colorado Memory Systems tape drives (DJ10 and DJ20). The external kit which we provide as item R-TD-K10 includes the external adaptor for the drive. This is a small board which uses one rear mounting but does not use a slot (i.e. it needs nothing from the main board bus). The adaptor board is available separately as part R-TD-A11.

There are two reasons for using the external adaptor: one is to move a single drive between two or more computers, each of which has the adaptor card; and the other is where you have no free 5.25" (or 3.5") mounting or floppy port available.

To move a tape drive between two machines, order one drive, one external kit, and one AB11 adaptor. Consider a few tapes as well - the 60 foot tapes are used with software compression to get the 120M/250M capacity.

Alternatively, for external use exclusively, one should give thought to using the Trakker; this is Colorado Memory System's tape device which connects up to the PC's standard parallel printer port.

LB/LB86 Print Bug

Fm MISOSYS, Inc. Wilbert Hannes reported a problem when attempting to print labels on a three-across grid simultaneously with three copies of each label. LB users are aware that with a single label print definition, you can request printing more than one copy of a label, or print on more than one label across. You can even specify both printing operations for the same print run.

When you print labels across a page (sometimes refered to as two-up or three-up) with but one image of each label, LB will print across the page for as many labels as you specify. If on the other hand, you specify one label across but more than one image of each label, LB will print a single image of each label, then re-print another set of labels for as many copies as you requested.

But if you specify both actions simultaneously, LB uses a separate print algorithm to print multiple copies sequentially with the labels across the page. This type of printing is the only specification which uses the particular algorithm. I cannot explain why it was designed that way as it was in the original LB 1.0 implementation. But a bug was introduced into the coding of the algorithm during the development of one of the 2.x upgrades.

In order to fix the bug, it was necessary to re-compile the MS-DOS version LB1.EXE module (or the LB/OV1 LS-DOS version module). If you have a need to do this kind of printing, you may return the disk which contains the LB1 file (should be disk 2 of 2) to obtain a corrected version.

RINGMOVE

Or how to re-order an indexed file with the least number of moves

by Roy Soltoff

Computers were originally designed to manage numeric calculations - the first being designed to aid in census data summarization. But for quite a few years, it seems that the manipulation of textual data has been the primary deployment of computers. Back in the Winter 89/90 issue of *The MISOSYS Quarterly* (Volume IV, Issue ii), I posted the results of a poll I came across which revealed that 98% of the preferred applications of PC users were in the areas of spreadsheet (numbers) and word processing (textual). 90% of users employed computers for database management (mostly textual), 84% for desktop publishing (textual again), and so forth.

With the solid emphasis on the manipulation of textual data, data sorting is a widely used operation. Sorting can also take a good deal of the computer resources - not so much because of the CPU power required for sorting, but for the time involved in shuffling data into and out of the computer's memory so string comparisons can be made. There are also many different sorting techniques, no one of which is the optimum for all given collections of data. Thus, employing the proper sorting algorithm should be done after a careful analysis of the organization of the data to be sorted.

Most of us, though, do not bother to write our own sort algorithms. We either use a sort function supplied by the host environment or adapt one found in a book to the environment we are using for our program which requires the sorting of data.

Whatever sort algorithm is ultimately chosen, there are only two distinct flavors used in adapting the sort algorithm to the intended application. The question is asked, Do I want to resequence the data? Or do I want to simply create an ordered index to the data set? Then again, just what do I mean by an ordered index?

Let me illustrate this by an example. Suppose you had a simplified set of data which consisted of five records, each containing surnames:

Smith
Jones
Lee
Adams
Clinton

If such a list of records were placed (ordered) alphabetically, it would appear as follows:

Adams
Clinton
Jones
Lee
Smith

It does not matter what sort algorithm was used to resequence the data; what matters is that now the data can be accessed sequentially in alphabetical order.

But suppose you did not want to change the physical sequence of the data? How could you achieve the same results (i.e. accessibility of the data in alphabetical order) without physically resequencing the data? Here's one such technique. Establish a secondary file which references the physical record numbers of the original set of data. This data set would now appear grouped as follows; for the purpose of this simplistic set of data, I'll just use a base index of 1.

1	Smith
2	Jones
3	Lee
4	Adams
5	Clinton

Thus, Smith is the first record; Jones is the second record, etc. Now before we can access any of the actual records of surnames, we have to first access the list of references. Whatever sort algorithm we employ to resequence the surnames alphabetically needs to be coded to alter not the sequence of the surnames, but the sequence of the reference numbers. Just like you refer to an index in the back of a book, the reference numbers become an index to the actual data. Let's look at the alphabetically-ordered data set when the index numbers are used.

4	Smith
5	Jones
2	Lee
3	Adams
1	Clinton

Well the set of index numbers doesn't look ordered; however, the index numbers taken in sequence now reference the data in a correct alphabetical sequence. For instance, if you want to print off the list, you read the first index number which is "4". This tells you that the first record is record 4, "Adams". The second record alphabetically is determined by the value of the second entry in the reference index; "5" points to "Clinton". Essentially, each record from 1 to n is addressed by RECORD[INDEX[J]]. Okay, so what have you gained? In this illustration, you have burdened yourself with having to read an extra piece of data before you can get at the data you want. True! In this short list, the payoff is not evident. But suppose your data set contained not just five records, each containing but a single surname, but the complete personnel data on every postal employee in the United States - that's about 800,000 records - each record of considerable size. Even with a small data set of ten thousand records - which fit very comfortably on a small micro, the time involved in data I/O for sorting purposes can become significant - more than stepping out for that cup of coffee. The use of

accessing data by means of an index can save considerable time when you have to re-sequence the data. And for most bases of data, data entry of new records and deletion of outdated records is a continuing process - one which requires the reordering of the record sequence for access. If you have less than 65536 records, you can even get by with an integer-sized index reference file.

Now that I have sold you on the usefulness of keeping data organized by using an index file, I must then expose you to one of the problems which can arise. Sooner or later, you will come up with a reason for wanting the data physically re-ordered. With the desired sequence stored in your index file, what technique can you employ to move all the records around? Obviously, the operation is a *moving* operation. Here's where I am finally getting around to the meat of the topic.

Many moons ago I worked for AT&T Long Lines in the microwave radio frequency interference group. The function of this group was to study the impact of proposed microwave routes (then used for long distance telephone and private line communications) on existing AT&T microwave repeater stations. Part of the work involved connecting together a sequence of topographic maps to examine the topography over the line of sight transmission route, noting the evidence of objects or terrain which could interfere with the proper transmission and reception of the microwave signal. Essentially, an elevation profile was developed and fed into a computer program which calculated the signal loss caused by the distance between the repeaters and the terrain profile. Another program was also used which analyzed all other microwave communications in the same frequency band which then calculated the level of competing signals reaching the repeater's antenna. The work was done to gain evidence in case a proposed transmitter would interfere with the reception of an existing station requiring a protest of the FCC application, or to plan a new route.

IBM 360 computers were used in a time sharing environment to access the data and programs. At the time, the particular time share system used was *MUSIC*, which stood for McGill University System for Interactive Computing. All of our programs were coded in FORTRAN, which is where I started out with my programming endeavors. Now IBM was never known for the quickest sorting implementations. Frustration with IBM's sort is probably what spawned many competing firms which specialized in mainframe sort implementations. Whether or not it was frustration or curiosity, there was a fellow by the name of Guy Ohlinger in the other District of our Division who began to implement his own sort routine which he called SRTMRK - short for SORT MARK (FORTRAN limited sup-program names to six characters at the time). SRTMRK was able to sort all FORTRAN data types - and I believe faster than the supplied mainframe sort utility. I recollect that Guy eventually re-coded it in 360 assembly and convinced the Accounting Department (back then, it was the Accounting Departments which had control of all computer installations) to install his SRTMRK as a company-wide resource. I am sure that was a tough sell since when did Accounting ever listen to Engineering?

SRTMRK was so-called because it did not re-sequence the data, but rather marked the data's ordered position in an index array. This is where you typically gain speed, not having to reshuffle all the data. Well since the need always arises that data some times needs to be physically rearranged, Guy also did another implementation of his SRTMRK which optionally re-arranged the data; this one was called SRTREA - short for SORT and REARRANGE. The re-arranging function used a technique which he called a ring move. I have never seen that term used in any textbook; however, it is a very elegant method of re-arranging an indexed sequence of data always guaranteeing the minimum amount of disk I/O. Let's look at that concept.

Given a set of data records whose ordered sequence is known, the optimum algorithm for re-arranging the data to the desired sequence is one in which no record is read or written more than once, and for which no record already in its proper position (by coincidence) is read at all. The ring move algorithm essentially treats the set of data and its corresponding index array as a linked list of disconnected record chains. At any record position, the algorithm needs to know simply what record goes into this position?

I have employed the ring move in two of my products. The oldest implementation was written in C as part of the PSORT utility which is included with PRO-WAM. The more recent implementation was done using assembler as a part of the HDPACK disk defragger. In the latter case, you really want to minimize the number of accesses to the disk in order to re-arrange the fragments of files throughout the disk. HDPACK actually sorts the disk files according to their hash index table entry numbers and their granule offsets considered relative to drive itself (i.e. more akin to the cluster number used in the directory structure of an MS-DOS disk). The ordered sequencing is performed using a shell sort, then the actual disk granules are re-ordered using a ring move. Using such a move, no disk granule is read or written more than once (i.e. one read, and one write).

PSORT uses a ring move to re-order the actual records of the targeted data file. Since PRO-WAM does not employ indexes to files but accesses files purely sequential, an external sort utility - such as PSORT - must re-sequence the data records. It is faster for PSORT to sort the records using an index, then re-sequence the records but once using the ring move.

Since this issue continues discussion of the C language, I thought it appropriate to illustrate a ring move implementation using PSORT. Besides, it is easier to follow an algorithm illustrated in a high-level language from the same algorithm illustrated in a low-level language. It calls to mind a lesson back in the early 60's when

we (the class) were learning to code a hypothetical binary computer called *HYPOVAC*. This had about eleven instructions, all coded using a single 8-bit byte to access the instructions and memory. I recollect that the machine had perhaps but a handful of instructions: STORE, LOAD, ADD, CPL, etc. In any event, we were given a program listing and were asked what the program accomplished. It generated the first n entries in the list of prime numbers by striking out every other, then every third, then every fourth, and so forth. But figuring that out was not intuitively obvious as the extremely low-level coding masked the actual algorithm. But back to the ring move algorithm.

PSORT was designed to sort various PRO-WAM data files - each of which had a different organization. Thus, the first full disk record of nearly all of PRO-WAM's data files (those sortable by PSORT) contain header data which provides PSORT with information on the record size, where the key field(s) to sort are located within the record, the size of the key field, and the type of the field: string or integer. This article does not attempt to cover the entire PSORT utility program, although the entire program is illustrated. I want to concentrate solely on the ring move algorithm. Suffice to say that the first portion of PSORT reads and validates the key field data contained in the header record. PSORT then allocates memory for two file buffers, an index array, and arrays for pointers to the key field data and the key data itself. The index arrays are used during the sorting process to keep the data position intact and use an index array to maintain the re-ordered sequence.

An initialization routine then prepares the index arrays. In particular, the array of index numbers used in the ringmove algorithm is initially filled with the numeric sequence 1, 2, 3,... by the code:

```
for(i=0; i < nrec; i++)
        {
        psub[i] = i;
        }
```

```
/* psort/ccc - 07/07/87
 * utility to sort PRO-WAM data files
 * change log
 *  02/18/86 - added code to support key_loc <> 0
 *  07/17/86 - Added code to fix pack bug if nrec >256
 *  05/04/87 - Added code to deal with integer and
masked keys
 *  06/08/93 - Moved PSORT22A patch code (1987) into
source
 */
#include stdio.h
#include string.h
#include fcntl.h
#include sgtty.h
#option REDIRECT OFF
#option INLIB
int i, j,     /* index counter */
    itemp,
    pack,     /* Set TRUE  on option to PACK deleted
records */
    rmndr, /* used to calculate record position in file
*/
    fd,    /* data file descriptor */
    block, /* 512-byte block number of record */
    offset,   /* offset within block */
    lrl,      /* logical record length of a record */
    nrec,     /* total number of records */
    keylen,   /* total key length after unpacking */
    first_rec,    /* 1st record of LRL treated as data */
    pos=0, /* offset for position for key_loc */
    *psub; /* array of subscripts to *pkeys array */
char   name[32], /* name of file to sort */
    *pt,      /* tempy char pointer */
    *buf1, /* one of two record I/O buffers */
    *buf2, /* the other one */
    *pkeyarray,  /* array of pointers to keys */
    *pkeys;   /* storage for key strings */
char *types[2] = {"string","integer"};
FILE *fp;     /* file pointer for fd */
union {
    char cbuf[256];     /* sector buffer */
    int  ibuf[128];
    } buf256;
struct {
    int loc;  /* location in record of sort key */
    int len;  /* length of sort key */
    int type; /* type 0=char, 1=int */
    int mask; /* mask, 0=no mask */
    } keys[3];
struct sgttyb sg;
extern int strcmp();
extern char *addext();
main(argc,argv)
int argc;
char *argv[];
{
    puts("\nPSORT Version 2.3 - PRO-WAM Data File Sort
Utility\n"\
    "Copyright (c) 1987 MISOSYS, Inc., All rights
reserved\n\n");
    ioctl(STDIN,TIOCGETP,&sg);
    sg.sg_control |= IO_BREAK;
    ioctl(STDIN,TIOCSETP,&sg);
    if ( argc > 3 ) usageerror();
```

The keys are then read into the allocated memory by the readkeys() function. The keys are then ordered by the shell() sort function with the psub array containing the resulting sorted order. Note that during the sort, the keys are addressed by a notation of the form, `v[index[j]]`. This is identical to the record access mentioned previously in our surname example.

The ringmove algorithm then works as follows:

1. `i = next = 0;`

   The variable "i" is used as a subscript of the index array - the array which holds the ordered sequence of the data. This is set to zero to start with the first position as C uses a base of 0 for all arrays. The variable "next" is used to keep track of the value of "i" when a chain of linked records is *broken*. The meaning of *broken* is really a condition of FALSE in step 3. "next" is also initialized to zero.

2. `while (i <nrec) {}`

   The variable "nrec" contains the quantity of records. The algorithm must proceed to examine all records in turn; the completion is noted when the subscript reaches the highest numbered record.

3.
```
if (index[i] == i || index[i]
== -1)
        {
        ++i;
        ++next;
        continue;
        }
```

   To use the ringmove algorithm, you must provide a method to be able to ascertain an instance of a record which has been moved into place. An implementation of ringmove is not bound to any particular method; rather it is the whim of the designer. In this implementation, I chose to use an index value of "-1" to indicate a record which has already been moved into

```
    if ( argc > 1 )
        {
        strcpy(name,argv[1]);
        if (argc == 2)
            pack = FALSE;
        else
            {
            lower(argv[2]);
            if (*argv[2] == '(')
                ++argv[2];
            if (strcmp(argv[2],"p") == 0 ||
strcmp(argv[2],"pack") == 0)
                pack = TRUE;
            else
                usageerror();
            }
        }
    else
        {
        puts("Enter name of file you wish to sort > ");
        if (!fgets(name,28,stdin))
            exit(-1);
        *strrchr(name, '\n') = '\0';/* overwrite the NL
*/
        puts("Do you wish to pack the file <Y,N> ? ");
        i = getchar();
        putchar('\n');
        pack = tolower(i) == 'y' ? TRUE : FALSE;
        }
/*
 * add '/DAT' file extension; open file for read/write
 */
    if (( fd = open( addext(name,"DAT"), O_RDWR)}==EOF )
        open_error(name);
/*
 * Read the header record
 */
    if ((read( fd, buf256.cbuf, 256)) != 256)
        {
        printf("Error in reading %s\n",name);
        exit(-1);
        }
/*
 * Extract info on nrec, lrl, and first data record
 */
    nrec = buf256.cbuf[1] * 256 + buf256.cbuf[2];
    lrl = buf256.ibuf[0x08];
    first_rec = buf256.ibuf[0x09];
/*
 * Get the info on the first two keys
 */
    keys[0].loc = buf256.ibuf[0x0a];
    keys[0].len = buf256.cbuf[0x16];
    keys[0].type = buf256.cbuf[0x17];
    keys[0].mask = buf256.ibuf[0x0c];
    keys[1].loc = buf256.ibuf[0x0d];
    keys[1].len = buf256.cbuf[0x1c];
    keys[1].type = buf256.cbuf[0x1d];
    keys[1].mask = buf256.ibuf[0x0f];
/*
 *  validity check
 */
    printf("Statistics: Nrec=%d, Lrl=%d\n",nrec,lrl);
    for (i=0; i<2; i++)
```

```
        if (keys[i].len)
          printf("  Key(%d): loc=%d, len=%d, type=%s,
mask=%04x\n",
i,keys[i].loc,keys[i].len,types[keys[i].type],keys[i].mask);
    if (nrec < 2 || nrec > 10000)
        {  printf("Can't sort %d records\n",nrec);
           exit(0);
        }
    if (lrl < 16 || lrl > 1024)
        {
        puts("File has invalid logical record length\n");
        exit(-1);
        }
/*
 * Check keys for validity
 */
    for (keylen=i=0; i<2; i++)
        {
        switch (keys[i].type)
            {
            case 0:
                keylen+=keys[i].len;
                break;

            case 1:/* unsigned int unpacked to 5 chars */
                keylen+=5;
                keys[i].len=5;  /* Reset key len to 5 */
                break;

            default:
                puts("Invalid sort key type\n");
                exit(-1);
            }
        if ((keys[i].loc + 1 >= lrl) || keys[i].loc +
keys[i].len > lrl)
            {
            puts("Sort key extends past end of
record\n");
            exit(-1);
            }
        }
/*
 * We need to allocate the following:
 * 2 disk I/O buffers of size lrl, [buf1 and buf2]
 * 1 pointer to integer array of length nrec*2, [psub]
 * 1 pointer to char array of length nrec*2,
[pkeyarray]
 * 1 character key array of length nrec * (keylen+1).
[pkeys]
 */
    if ((pkeys=alloc(lrl*2 + nrec * (keylen+4))) == 0)
        {
        puts("Can't allocate storage for keys\n");
        exit(-1);
        }
    buf1 = pkeys; pkeys += lrl;
    buf2 = pkeys; pkeys += lrl;
    psub = pkeys; pkeys = pkeys + nrec * 2;
    pkeyarray = pkeys; pkeys = pkeys + nrec * 2;
    init(psub,pkeyarray,pkeys);  /* Initialize index and
key pointer arrays */
/*
 * Kludge to ensure ERN offset is zero!
 */
```

the correct position. Therefore, this block of code steps through the index array examining (1) if a record is coincidentally in the correct position and does not need to be moved, or (2) has already been moved. If either situation is TRUE, the subscript (and the *next* tracker) are incremented and the algorithm continues with step 2 - the outer *while* loop. If neither are TRUE, then a record needs to be moved; the algorithm proceeds to step 4.

4. `readrec(i,buf2,lrl); link=index[i]; index[i]=-1;`

The readrec() function reads the record referenced by the first argument. The record is read into the buffer identified by the second argument. So note that here we are reading the ith record which is not the record which belongs at this position but the record at this position. To clarify this, lets look back at our simplified case of surnames. Step 3 would find that neither the record is in place or the end of a chain is reached; thus, step 4 would be exercised with i equal to 0. The "Smith" record would be read into buffer 2. The record linking to this position would be the record which belongs here or the record pointed to by the index (`index[i]`). Step 5 is then exercised.

5.
```
while ((index[link]) != -1)
 {
 readrec(link,buf1,lrl);
 writerec(i,buf1,lrl);
 index[i] = -1;
 i = link;
 link = index[i];
 }
```

In this step of the ringmove, the code block traverses the records which are linked together until the end of the chain is reached (i.e. the completion code of -1 is found). The "while" tests for that condition.

The code block reads a linked record

into buffer 1. In our surname example, that would first be the fourth record, "Adams"; 4 being the value of index[0]. Buffer 1 now has the record which should be in position i (i=0). That record is written to its proper place, the index value is changed to the completion code, the index "i" is changed to the link value (4), and a new link is ascertained by the index value at index[4]. The new link is 3. Why "3" you may ask? Well since we have just moved "Adam" to its correct position, we need to read the record which belongs to the position just vacated by "Adam". That's record 3!

The next time through the code block, record 3 ("Lee") is read, written to position 4, the index at 4 is changed to the completion code, the subscript is changed to 3, the link is changed to 2. The *while* tests index[2] and finds a 5 - nothing to terminate the block so the code block is re-executed.

The next time through the block, we read Jones from 2 and write Jones to 3, designate position 3 as complete, change the subscript to 2 and the link to 5. Once more through the code block.

The next time through, we read Clinton from 5 and write Clinton to 2, show index[5] as complete, change the subscript to 5 and the link to index[5] which is equal to 1. The *while* now tests index[link] and finds that it is equal to -1, a completion code. Therefore, the *while* terminates its code block. Proceed to step 6.

6. `writerec(i,buf2,lrl); i = next;`

The record we originally read into buffer 2 is now written into position 5, the current value of the subscript. The subscript is now changed to the value of next and the code execution continues at the original *while* test in step 2. From there, the code in step 3 finds that all records have been moved. The

```
    fp = fpup(fd);      /* get file pointer */
    *(*fp+9) = '\0';    /* set EOF offset to zero */
/*
 * read the keys into the storage set up for them
 */
    readkeys(pkeyarray);

/*
 * Sort the keys via the index array
 */
    shell(psub,pkeyarray,strcmp);
/*
 * re-order the data file records via a ring move
 */
    ringmove(psub);
/*
 * Pack the file if requested
 */
    if (pack)
        packfile();

    close(fd);
}

/* routine to position the file to a given record */
position(recno)
int recno;
{
    lseek( fd, (long)recno * (long)lrl + (long)pos, 0);
}

open_error(name)
char *name;
{
    printf("Can't open %s\n",name); exit(-1);
}

readrec(recno,buf,len)
int recno, len; char *buf;
{   position(recno + first_rec);
    if ((read( fd, buf, len)) != len)
        {
        printf("Data file read error, record
%d\n",recno);
        exit(-1);
        }
}

writerec(recno,buf,len)
int recno, len; char *buf;
{   position(recno + first_rec);
    if (( write( fd, buf, len)) != len)
        {
        puts("Data file write error\n");
        exit(-1);
        }
}

shell(index,v) /* sort v[index[0]]...v[index[n-1]] */
int index[]; char *v[];
{
    int gap, j, temp;
    puts("Sorting key fields\n");
/* No longer using Knuth's recommendation - doesn't
```

```
work for nrec < 4 */
    for (gap=nrec/2; gap > 0; gap /=2)
        for (i=gap; i<nrec; i++)
            for (j=i-gap;
        j>=0&&(strcmp(v[index[j]],v[index[j+gap]])>0);
            j-=gap)
                {
                temp=index[j];
                index[j]=index[j+gap];
                index[j+gap]=temp;
                }
}

init(psub,pkeya,pkey)
int psub[]; char *pkeya[], *pkey;
{
    for(i=0; i < nrec; i++)
        {
        psub[i] = i; /* init integer subscript array */
        pkeya[i] = pkey++; /* initialize the keys
pointer array
*/
        pkey += keylen;
        }
}

readkeys(pkeys)
char *pkeys[];
{
    puts("Reading key fields\n");
    for (i = 0; i < nrec; i++)
        {
        pt = pkeys[i];
        for (j=0; j<2; j++)
            if (keys[j].len)
                { /* set position offset to key loc'n */
                pos = keys[j].loc;
                switch (keys[j].type)
                    {
                    case 0:
                        readrec(i, pt, keys[j].len);
                        *(pt+keys[j].len)='\0';
                        if (keys[j].mask)
                            while (*pt)
                                *pt++&=keys[j].mask;
                        else
                            pt+=keys[j].len;
                        break;
                    case 1:
                        readrec(i, &itemp, sizeof(itemp));
                        if (keys[j].mask)
                            itemp&=keys[j].mask;
                        sprintf(pt,"%05u\0",itemp);
                        pt+=5;
                        break;
                    }
                }
        }
    pos = 0;
}
```

use of next ensures that all disconnected record chains are dealt with.

Note that every record was read but once, and written but once. This satisfies the requirement of optimality.

As a final submision to the subject matter of sorting, I am reprinting an updated version of my SEESHELL program which was useful for visually demonstrating the underlying behavior of the shell sort. SEESHELL first appeared in NOTES FROM MISOSYS, Issue 3, July 1984. The program used video memory as the data memory - each memory address contained a character of data. Thus the progress of the sort could be observed simply by looking at the screen.

SEESHELL was originally written to be compiled by LC or PRO-LC. Since my MC compiler has been out for many years, I decided to bring SEESHELL up to date by changing some of its structure for MC's use, and to allow direct compilation for either Model III or Model 4 targets. SEESHELL accesses video memory directly which requires that HIGH$ be below X'F400'. As such, the code incorporates testing for a usable value of the Model 4 high memory pointer to ensure that your computer does not crash.

Note that SEESHELL uses a single precision floating point function, frnd(), which requires use of the "+f" compiler option. The companion DISK NOTES to this issue contains the program source as well as the Job Control Language file used to compile SEESHELL.

```
ringmove(index)
int index[];
{  int link, next;

   puts("Rearranging data file\n");
   i=next=0; /* init to start the ring
move */
   while (i < nrec)
      {  /* is record already in
place? */
      if (index[i] == i || index[i] ==
-1)
         {
         ++i;
         ++next;      /* bump to next
record */
         continue;    /* and continue
with while */
         }
      readrec(i,buf2,lrl);   /* read
record at this index */
      link = index[i];    /* establish
link to next */
      index[i] = -1;      /* show end
of link done */
      while ((index[link]) != -1)   /*
do until end of link */
         {
         readrec(link,buf1,lrl);   /*
read record belonging here */
         writerec(i,buf1,lrl);  /*
write proper record here */
         index[i] = -1;      /* show
this is moved */
         i = link;           /* advance
the link */
         link = index[i];
         }
      writerec(i,buf2,lrl);      /*
write end of link */
      i = next;     /* restart to
search for next record out of place */
      }
}

usageerror()
{
    puts("Usage is: PSORT datafile
[PACK]\n");
    exit(-1);
}

lower(str)
char *str;
{  char c;
   while (c = *str)
      *str++ = tolower(c);
}

packfile()
{
   puts("Packing data file\n");
   i = nrec;
   while (--i)          /* read file in
reverse offset from 0 */
      {
      readrec(i, buf1, lrl);
      if (*buf1 != 255)
         break; /* exit when un-deleted
record found */
      }
   if (++i != nrec)    /* change nrec
if any deleted are packed */
      {
      fp = fpup(fd);          /* get
file pointer */
      *(*fp+14) = *(*fp+12);     /* set
ERN to NRN */
      *(*fp+13) = *(*fp+11);
      seek(fd,0,0);
      read(fd, buf256.cbuf,256);    /*
Reset nrec */
      buf256.cbuf[1] = (i & 0xff00) >>
8;
      buf256.cbuf[2] = i & 0x00ff;
      seek(fd,0,0);
      write(fd, buf256.cbuf, 256);
      *(*fp+2) &= 0xbf;          /*
reset POSN bit */
      }
}
/* provide patch space */
#asm
   DC 256,0
#endasm
```

```
/* seeshell/ccc by  Roy Soltoff
 * Updated for MC/PRO-MC compilation
 * and conditional code for Model III
or 4
 */
#include stdio.h
#include math.h
#include z80regs.h
#option args OFF
#option redirect OFF
#option maxfiles 0
#option INLIB   /* needed for call() */
#ifdef dos6
#define CRT 0xf800      /* origin of
Model 4 CRT memory */
#define SIZE 1920
union REGS reg;
#else
#define CRT 0x3c00      /* origin of
Model III CRT memory */
#define SIZE 1024
#endif

main()
{
   char *v;
   float v1, v2;
   int i, c, ascend(), decend();
   /* CRT address and screen size */
```

```
    v = CRT;
    cls();
#ifdef dos6
    reg.HL = reg.B = 0;
    (void) call(100, &reg);       /* get
HIGH$ */
    if ( reg.HL > 0xf3ff)      /* if
higher than X'F3FF' */
        {                      /* abort */
        puts("HIGH$ must be below
X'F400'\n");
        exit(-1);
        }
    else
        {
/* Use following ASM code for Model 4
only */
#asm
    DI              ; interrupts off
    LD A,(78H)      ; get memory map
    AND 0FCH        ; strip all but SEL0,1
    OR 2            ; swap in video memory
    OUT (84H),A     ; select new memory map
#endasm
        }
#endif
    v2 = 94; /* set random upper */
    for (i=0; i<SIZE; ++i)
        /* set char 32 - 126 */
        *v++ = (char) (frnd(94.) + 32);
    shell(CRT, SIZE, ascend );
    for (i=0; i<10000; i++);
    shell(CRT, SIZE, decend );
    c =getchar();
    cls();
    exit(0);
}
shell(v, n, order) /* sort v[0]... v[n
-1] */
char v[];
int n, (*order)();
{
    int gap, i, j, temp;
    /* set gap as recomended in Knuth,
Vol , po 95 */
    for (gap = 1; gap <n ;gap
=(3*gap)+1)
        ;
    /* set gap larger than n, then back
up two steps */
    gap = (-gap)/3;
    gap = (-gap)/3;
    /* now gap has been initialized */
    for ( ; gap > 0; gap = ( -gap)/3 )
        for (i=gap; i<n; i++)
            for ( j=i-gap; j>=0 && (*or-
der)
(v[j],v[j+gap]); j-=gap)
            {
            temp=v[j];        /* swap
elements */
            v[j]=v[j+gap];
            v[j+gap]=temp;
            }
}

cls()
{
    puts("\x1c\x1f");
}

ascend(v1,v2)
char v1,v2;
{
    return (v1 > v2) ? TRUE : FALSE;
}

decend(v1,v2)
char v1,v2;
{
    return (v2 > v1) ? TRUE : FALSE;
}
```

```
/*
 * Function getbuff()
 */

char * getbuff(num)
    int num;
{
    unsigned tempsize;
    char *buffer;
    unsigned segment, offset;
    long address;

    for (;;)
    {
    if (!(buffer= (char
*)malloc(buffsize)))
        return buffer;   /* error if no
more memory */

    /* calculate full 20-bit buffer
address */
    address = (offset=FP_OFF((char
far*)buffer) + (FP_SEG((char
far*)buffer)<<4));

/* strip page address and ensure size
does not wrap */
    if (((address & 0xFFFF) + buffsize)
< 0xFFFF)
        return buffer;

    /* free up this buffer and allocate
the dummy */
    free(buffer);
    if (dummy[num])
        free(dummy[num]);
    tempsize = 0x10000 - offset;
    if (!(dummy[num] = (char
*)malloc(tempsize)))
        return buffer;   /* error if no
more memory */
    }
}
```

UPGAT/CMD Ver. 2.0

By Scott Toenniessen 1991

Some older disk directory repair utilities such as PFIX/CMD from the Toolbox for LDOS 5 and Toolbelt for LS-DOS 6 by Powersoft (both now sold by MISOSYS) are unable to correctly repair the GAT sector found on LDOS 5.3.x and LS-DOS 6.3.x type disks. This occurs because beginning with LS-DOS 6.3.0, the previously unused bit 3 of byte CDh in the GAT sector was assigned to indicate whether or not a disk used the new or old style dating. When this bit is set, 6.3 style dating is assumed, when it is reset, pre-6.3 style dating is indicated.

Older versions of directory repair utilities (specifically PFIX/CMD) reset this bit when repairing a disk, even on 6.3 style disks. This causes LS-DOS 6.3.x to think it is reading an old date style disk and causes it to omit the time stamp and incorrectly interpret the date stamp when using the DIR command. Running DATECONV sets the bit to 1, but since the dates were already in the 6.3.x format, they are scrambled and the time fields are set to 12:00.

This is where UPGAT/CMD comes in. This Model 1/3/4 program simply sets GAT+CDh bit 3 to 1. This allows LDOS/LS-DOS to correctly recognize the dating style used on the disk.

Operation: After a directory repair operation which corrects a corrupted GAT sector (PFIX :D,G would do that), simply type:

```
UPGAT :d
```

where d is the drive number. The colon may be omitted. Optionally, the command:

```
UPGAT :d (old)
```

may be executed where old indicates that pre-6.3.x style dating is used. This option is included in case the program is ever used on an earlier LDOS/LS-DOS disk by mistake.

Comments: In my experience with PFIX6/CMD, I only need to run UPGAT after

```
;*****************************************************************
;UPGAT v 2.0 (c) Scott Toenniessen 1991
;
;For use with pfix/cmd and pfix6/cmd GAT repairs
;Update disk configuration byte CD bit 3 to indicate
;LS-DOS 6.3.x style dating
;
;Execute via:  upgat :d (old)  where d is the drive
;number and old is an optional param. to set pre 6.3.x
; dating. The colon may be omitted
;
;This version of upgat is may be freely distributed as
;long as the authors name and copyright notice is left
; intact.
;It may be modified as long as the above is observed
;
;Compile with: MRAS upgat/asm -gc    for LS-DOS 6
;              MRAS upgat/asm -gc (p1) for LDOS 5
;
;*****************************************************************
DOS5   EQU @@1
DOS6   EQU .NOT.DOS5
   IF DOS6
*GET   SVC
   ENDIF
   IF DOS5
*GET   EQUATES
   ENDIF
   ORG 7000H
START  PUSH  HL        ;Save command line arg
HELLO  LD HL,HELLO1    ;Display greeting
   SVC @DSPLY
   LD HL,HELLO2
   SVC @DSPLY
   LD HL,HELLO3
   SVC @DSPLY
   LD HL,LF
   SVC @DSPLY
ARG1   POP HL          ;Get command line arg
   LD A,003AH          ;Check for ':'
   CPI
   JR Z,ARG2
   DEC HL ;if no colon, check for drive number anyway
ARG2   LD A,000DH      ;check for carriage return
   CPI
   JR Z,ERR2           ;  if CR, quit
   DEC HL         ;else check for number 0-7
ARG3   LD A,0038H      ;check for 8
   CPI
   JR Z,ERR3       ;if 9, go error
   DEC HL
   LD A,0039H          ;check for 9
   CPI
   JR Z,ERR3       ; if 9, go error
```

```
    DEC HL
    LD C,(HL)     ;Specify drive number
    PUSH  HL          ;Save HL PARM
    PUSH  HL          ;Save HL for WRGAT
    SVC @CKDRV ;Check drive availability
    JR NZ,NODISK      ;    go if not ready
RDGAT  LD E,00H       ;Read dir sector 0
    LD HL,BUFFER      ;    (GAT sector)
    IF DOS6
    SVC @RDSSC
    ENDIF
    IF DOS5
    SVC RDSSEC
    ENDIF
    JR NZ,ERR1    ;Go if error reading
PARM  POP HL          ;Recall HL
    INC HL     ;Advance past drive spec
    PUSH  DE          ;Save dir cyl.
    LD DE,PRMTBL      ;Point to parameter
table
    SVC @PARAM        ;Parse parameters
    JR NZ,ERR2        ;Go if parm. error

OPARM  LD BC,0        ;Set default to off
    LD A,B    ;Merge the hi and lo
orders
    OR C
    JR Z,SETBIT       ;If old=n, go

RSTBIT LD HL,DSKCFG ;reset CDH bit 3
for pre LS-DOS 6.3 dating
    RES 3,(HL)
    JP WRGAT

SETBIT LD HL,DSKCFG     ;set CDH bit 3
    SET 3,(HL)    ;LS-DOS 6.3.x dating

WRGAT  LD E,00H  ;Save modified GAT
sector
    POP DE          ;Get dir cyl
    POP HL          ;Get drive number
    LD C,(HL)
    LD HL,BUFFER      ;Point to buffer
    IF DOS6
    SVC @WRSSC        ;Save
    ENDIF
    IF DOS5
    SVC WRPROT
    ENDIF
    JR NZ,ERR1    ;Go if error writing
    JP EXIT       ;else exit to LS-DOS

ERR1   LD C,A     ;@RDSSEC/@WRSSEC error
    SET 6,C    ; set short error message
    SVC @ERROR        ;    display error
and exit to LS-DOS
ERR2   LD HL,SPECMSG        ;Command line
spec error
    SVC @DSPLY    ; Display error message
    SVC @ABORT        ;  abort to LS-DOS
ERR3   LD HL,ILLMSG     ;Illegal drive
number
    SVC @DSPLY
    SVC @ABORT
```

```
NODISK LD HL,DSKMSG     ;Drive not ready
error
    SVC @DSPLY    ; display error message
    SVC @ABORT    ; abort to LS-DOS

EXIT   LD HL,DSKCFG     ;Exit to LS-DOS
    BIT 3,(HL)        ;  Check which
messageto use
    JR Z,BYE62
BYE63  LD HL,BYE1    ;Use 6.3.x message
    SVC @DSPLY
    JP LSTJP
BYE62  LD HL,BYE2    ;Use pre 6.3.x
message
    SVC @DSPLY
LSTJP  LD HL,0000H      ;    no error
    SVC @EXIT         ;  EXIT

DSKCFG EQU $+00CDH          ;locate disk
configuration byte in GAT
BUFFER DS 256
DSKMSG DB  'Drive not available',0DH
SPECMSG DB    'Must invoke via: upgat :d
where d is the drive to fix',0DH
ILLMSG DB  'Illegal drive number',0DH
HELLO1 DB  'UPGAT v. 2.0 (c) 1991 by
Scott Toenniessen',0DH
    IF DOS6
HELLO2 DB  'Updates GAT sector of LS-DOS
6.3.x formatted disks',0DH
HELLO3 DB  'after being repaired with
PFIX6/CMD',0DH
    ENDIF
    IF DOS5
HELLO2 DB  'Updates GAT sector of LDOS
5.3.x formatted disks',0DH
HELLO3 DB  'after being repaired with
PFIX/CMD',0DH
    ENDIF
LF  DB 0DH
    IF DOS6
BYE1   DB       'GAT is now 6.3.x
compatable!',0DH
BYE2   DB  'GAT is now pre 6.3.0
compatable!',0DH
    ENDIF
    IF DOS5
BYE1   DB  'GAT is now 5.3.x
compatable!',0DH
BYE2   DB  'GAT is now pre 5.3.0
compatable',0DH
    ENDIF

PRMTBL DB  'OLD    '      ;Old style dat-
ing parameter
    DW  OPARM+1
    DB  'O      '
    DW  OPARM+1
    NOP           ;End of table

    DATE
    TIME
    END START
```

PC DMA Transfer

or

How to work around DMA buffer origins from a high-level language

by Roy Soltoff

Anyone who has programmed a PC at a low level for any number of years may have come upon one problem unique to Direct Memory Access (DMA). Material has already appeared in *The MISOSYS Quarterly* concerning the use of DMA on the Model 4 equipped with an XLR8er card (see *The Final Solution to the XLR8er Question* issue VI.i; *How to Roll Your Own on the XLR8er*, issue V.iv). This article will therefore present only a specific difficulty when using DMA on a PC (i.e. x86-based computer).

The dominate x86-architecture chip currently is probably the Intel 486. This chip with 32 address lines, can access gigabytes of memory; however, if one is using MS-DOS as the operating system, the CPU architecture reverts to the 8086 memory structure with 20 address lines capable of directly addressing 1 megabyte of RAM. The utilization of the 1 megabyte was defined by IBM when their first PC was being designed. This memory structure provided the first 640 kilobytes for user memory, with the rest of addressable memory used for video memory and the Basic Input Output System (BIOS). Memory banking schemes no different than that used on the TRS-80 Model 4 provide a means of access to additional memory. However, for the purposes of program execution, they must run within the 640K region.

Although this appears like a large amount of memory, it is not. As all systems grow, we soon exceed our limits.

Mostly anyone who has puttered in the MS-DOS arena knows that the memory region is not really a linear address space when any x86-architecture CPU is being operated in its 8086 mode. When the 16-bit CPUs first arrived after the 8080, Z80, 6805, 6800, etc. chips, there were three major manufacturers - each with a chip designed under a different memory architecture. Some folks say that Motorola, with its purely linearly addressed 68000 had the best design. Apple seemed to think so. So did the workstation designers who used that chip to design Unix-based machines. Zilog had a Z8000 chip, intended to capitalize on their success with the Z80. The Z8000 used a segmented memory design with a 64K segment; the segments were addressed on 64K boundaries. Thus an absolute address could be generated by using a segment register to address high-order memory lines with the program counter addressing the address within the 64K segment. Intel also designed their 8088 and 8086 chip with a segmented architecture using 64K segments; however, unlike the Zilog chip, Intel's used a segment origin every 16 bytes. There's segment registers whose contents must be shifted by 4 bits then added to the program counter address (referenced as a *segment offset*) in order to present a 20-bit address to the CPU bus. This certainly appears to be a *Mickey Mouse* scheme. The one thing which Intel had going for them was their creation of a low-cost 8088 CPU which had but an 8-bit internal bus and the 20 address lines. This was cheaper than the 8086 with an internal 16-bit bus as well as cheaper than the 68000 and Zilog Z8000.

According to most text's covering the design of IBM's PC, the lower cost factor of the 8088 was a primary reason behind their choice of the Intel chip. The rest is history; IBM's selection with a resultant industry built around IBM's choice made Intel's the chip of choice. The Z8000 fell by the way side. Motorola still maintained a presence with Apple's use of the 68000 for the Macintosh, and the workstation crowd building Unix boxes around the 68000. Also, as Intel created the 80286, 80386, and 80486, Motorola continued with the 68020, 68030, and 68040.

Let's briefly review the memory addressability of the Intel 8086 CPU family. These processors use a 20-bit address bus; thus, addressable memory is from absolute address X'00000' through X'FFFFF' - a total of 1,048,576 bytes.

Since the 8086 incorporates only 16-bit registers, memory space is accessed using a segmented memory scheme. The entire 1 Megabyte address space is divided into 65536 segments numbered from X'0000' through X'FFFF'. Each segment begins at a 16-byte boundary starting from address X'0000'. The following table illustrates the first few and last few segment addresses mapped into the one megabyte absolute address space.

Segment Number	Address Origin
X'0000'	X'00000'
X'0001'	X'00010'
X'0002'	X'00020'
X'FFFD'	X'FFFD0'
X'FFFE'	X'FFFE0'
X'FFFF'	X'FFFF0'

An absolute address is constructed by adding an offset, also contained in a 16-bit register, to the address **origin** of a segment number (not the segment number itself). In actuality, since the segment is contained in a 16-bit register, the CPU internally shifts the segment number left by four bits then adds the offset value. This produces a 20-bit result - the absolute address.

Since the segment offset is also a 16-bit

value, it is a number between X'0000' and X'FFFF'. Obviously, when the segment is X'FFFF', an offset greater than X'000F' would be invalid since the address arithmetic algorithm just discussed would overflow a 20-bit value. Likewise, an offset value greater than X'7FFF' could not be used with a segment value of X'F800'. The 8086 CPU does not treat these conditions as errors but resolves the result to an address within its addressable range; thus; they would wrap past X'FFFFF' to X'00000'. It is a quirk that the 80286 had a error in its design structure such that when operating in 8086-mode, the address overflow would not wrap but actually result in an address above the 1 megabyte region. Extended memory managers have made use of this fact.

Because the segmented architecture references segment origins at 16-byte boundaries, another term has been used when discussing 8086 memory space. That term is a "paragraph". A paragraph is any 16-bytes of memory which begins at a 16-byte boundary. There are, therefore, 65536 paragraphs of memory. Thus, a segment may be considered to originate at every paragraph.

Because PC addresses use a summation of a shifted segment with an offset, there are many different combinations of segment and offset which produce identical 20-bit addresses! As a matter of fact, there must be 4K of combinations which produce any given 20-bit address. Let's look at an example to resolve and understand this. Address X'00040' can be referenced by any of five combinations of segment and offset (generally written as "segment:offset").

```
0000:0040
0001:0030
0002:0020
0003:0010
0004:0000
```

These may look oversimplified, so perhaps another example may illustrate the nature of segment:offset summations. The address X'6789A' can be generated from these handful of segment:offset values (as well as many other combinations:

```
6000:789A
6001:788A
5FE2:7A7A
```

In other words, since the segment value is shifted left by four bits prior to adding, a change of one in the segment value could be counterbalanced by an opposite change of 16 in the offset value and still sum to the same machine address.

Once an address is greater than 0000:FFFF, it can generally be realized using any of 16K combinations. In an actual 8088 or 8086, since addresses above X'FFFFF' wrap around to 0, any address can be realized via any of 16K combinations - even the first 64K of memory.

So why bring up all this history and addressing schemes? Well one reason has to do with the DMA chip used in the PC, and the means of programming it. You see that although the 8086 family of CPUs use a segmented architecture on *paragraph* boundaries (that's the term applied to the 16-byte segment address), the DMA chip handles a maximum of 64K at a time **but the memory references must be on 64K boundaries!**

The original PC used an Intel 8237A DMA controller. The address registers in this device are 16-bits wide; thus, the device can handle at most a transfer of 64K. However, PC design uses a twenty-bit address to select any byte in the one megabyte address space. For the purpose of transferring a twenty-bit address to the DMA controller and associated hardware, the PC provides a *page register* for specifying the upper four-bits of the twenty-bit address, and an *address register* for specifying the lower sixteen-bits of the twenty-bit address. The page register controls the 64K block of memory to be referenced in the DMA transfer while the address register passes an address within that 64K block to the DMA chip. Both of these registers are port mapped. This scheme is more akin to the segmented architecture of the Z8000 than that used in the x86 chips.

Enter DOS Memory Allocation

MS-DOS provides three service calls dealing with memory allocation which can be used by the programmer at a low-level: Allocate Memory - function call 48H; Free Allocated Memory - Function Call 49H; and Modify Allocated Memory Blocks - function call 4AH. The latter is used to grow or shrink the size of a memory block previously allocated. With these functions, the sense of *memory block* refers to a specified number of paragraphs; memory is allocated to a program in paragraphs - multiples of sixteen bytes. That is also to say that every memory allocation is provided on a paragraph boundary. Since a segment address is coincident with paragraph boundaries, every memory allocation can also be considered the start of a 64K segment.

Not every program running under MS-DOS is big - although at times it seems that way. If a given program requires less than 64K of data space, every data item can be stored in a single 64K of space referenced by a single segment address - only the offset need be unique. When a program needs a data space in excess of 64K, each data item referenced must be resolved by a possibly unique segment address and an offset address. Thus, 32 bits are needed to resolve all data addresses. Because of this, a program with a data space larger than 64K will run slightly slower than an identical one with less than 64K of data. The same is true for the program size; programs larger than 64K require 32-bit resolution for function calls.

Most programs for MS-DOS are written in a high-level language, C is typically the language of choice for commercial software; however for Windows, Visual BASIC is making some inroads. It is advantageous to be able to tell a compiler that a given program is to use nothing more than 64K of data space. This is done by predeclaring a memory model for the program. The Microsoft C compiler provides six memory models which can be pre-

declared. These are:

tiny	both code and data limited to a single 64K region.
small	code and data are limited to 64K regions each.
medium	data is limited to 64K but code is unlimited.
compact	code is limited to 64K but data is unlimited.
large	neither code nor data is limited but individual arrays are limited to 64K.
huge	same as large but individual arrays can exceed 64K.

If we consider but the first three models, all data addressing is within a single 64K block of memory. This means that the same segment address will be used for the access of all data - only the offset will be different. As previously shown, the segment origin will be a paragraph boundary with a one in 16K chance of being at a 64K boundary.

Now look at DMA operation

A DMA controller generally supports two types of data transfer: memory-to-memory, or memory to/from I/O device. In brief, the DMA chip is passed a base address and a word count - along with other specifications as to the type of transfer. Assuming data from an I/O device (such as a floppy data channel) is being read into memory, a byte is transferred to memory, the word count is decremented, and the address is incremented. This sequence will continue until the word count "rolls over" from X'0000' to X'FFFF'.

What would happen if an attempt was made to transfer 16K of data when the machine address origin was 46A0:2603? An offset of X'2603' looks harmless enough. However, this calculates to a machine address of X'4E003. This machine address value would require a page address of X'4' and a base address of X'E003'. After a transfer of one byte, the base address would be X'E004' with a count of X'3FFF'. After the transfer of 8189 bytes, the base address would be X'FFFF'. The next byte transferred would cause the address to wrap to X'0000'. Thus before even 8K was transferred, the base address would wrap from X'0000' and another almost 8K of bytes would be transferred not to where you thought would be, but to some other area overwriting other data or program code. This is called a DMA boundary error!

DMA was used in the original IBM PC to transfer data to and from the floppy disk controller. The Floppy Disk Controller (FDC) chip used in the PC allows multi-sector I/O. The BIOS also supports the ability to read or write more than one sector of data with a single service call. Allowing the FDC to handle multiple sector I/O can definitely speed up the transfer when you want to sequentially transfer more than one sector - such as in a track read or write operation.

A floppy disk sector under MS-DOS is 512 bytes. A 360K floppy has nine sectors per track - or 4.25K. Since the BIOS allows multiple sector transfers using a multi-sector memory buffer, the location of such a buffer must be entirely within the first 59.75K of a segment whose address is on a 64K boundary. An attempt to use DMA to transfer even a larger block of data would impose a greater restriction on the actual location of the memory buffer. To guard against DMA boundary errors associated with floppy I/O, the original IBM PC BIOS tested the resolved machine memory address to ensure that the buffer region remaining in the 64K-bounded segment block was as large as the number of bytes to be transferred. If the available memory was too small, the BIOS returned an "Attempt to DMA across 64K boundary" error and aborted the floppy I/O request.

I'm sure that this problem caused much consternation for a few years. Eventually, the programmers who wrote the BIOS code decided to have the BIOS break a transfer request which traversed a 64K boundary into multiple transfers automatically setting the proper values into the DMA registers; however, anyone directly programming the DMA chip would have to ensure that an overrun condition would not prevail. But to this day, there are still machines with a BIOS whose FDC DMA initilization code cannot handle a request which would traverse a 64K boundary; they will return a DMA boundary error.

Now enter high level language

I typically do not get involved with much low-level programming on the PC. But it does happen. Ever since MISOSYS acquired the Powersoft product line, we have duplicated TRSCROSS on a Mountain 3200 duplicator. This is a 5.25 inch drive contained within an apparatus using a hopper mechanism to feed diskettes to the drive under software control. Commands can be sent to the device to drop the engaged diskette into one of two bins. a bin for good disks and a bin for bad diskettes. The commands and duplicator status are accessed via a serial port; the floppy drive is accessed via a standard 34-pin floppy bus. The duplicator is therefore connectable to either a TRS-80 or a PC. And in fact, it is used with both computer types.

Powersoft used the Mountain 3200 for duplicating their TRS-80 diskettes; it was, in fact, Powersoft who wrote the TRS-80 duplicator code and licensed it to Mountain. Powersoft also wrote a short assembly language program to drive the duplicator from a Tandy 1000 so they could use it to duplicate TRSCROSS.

The PC BIOS provides floppy I/O service calls in a nature similar to the TRS-80. The FDC chip used in the PC is a little smarter than the chip used in the TRS-80 as the PC's chip handles the formatting pattern directly. All you need to provide it for formatting is a data table containing essentially the head, sector, and track numbers. It doesn't take a lot of code to write

a program which can format a disk, read from a source disk, then write to the formatted destination disk. The BIOS code can handle a large number of sectors transferred at one time (given the above attention to buffer origin relative to transfer size) - but only on single sided media. The BIOS does not automatically reference the second side when an attempt is made to transfer more than a track of sectors; the BIOS goes merrily along continuing with the same side. MS-DOS diskette organization is akin to LDOS and LS-DOS for 2-sided media; a sequence of sectors traverses all like-numbered tracks of a cylinder before stepping to the next higher-numbered cylinder. So for a 2-sided floppy disk, you can only transfer a track at a time.

I suspect that because of this, and because Powersoft did not need a lot of disk space for the TRSCROSS program, they decided to put it on single-sided media. Not only that, they used a disk structure which was prevalent on the first PCs - eight sectors per track rather than the then current nine. This all resulted in a quicker duplication time. Speed was important, also because the source diskette was a *master* floppy. Unlike my use of hard drive based diskDISK floppy diskette images to duplicate my TRS-80 products directly from a hard drive, reading the TRSCROSS source from a floppy was slower to access the source.

My problem was that I wanted to use the Mountain for duplicating some of my MS-DOS products - like LB86. The data base requires a much larger diskette capacity for releasing the product; it just fits on two 360K diskettes.

Enter 2FILE and 2FLOPPY

The March 1991 issue of *PC Magazine* contained a pair of utilities written by Stephen D. Cooper which could be useful for assisting in the modem transfer of a floppy disk, or in easily duplicating floppy disks on a one-floppy machine. One program, called 2FILE, read a floppy disk, detected its configuration, and created a file image - including a configuration header - of the floppy onto a hard drive. The companion program, 2FLOPPY, read the image from the hard drive, set up the DOS to configure the drive according to the configuration of the original floppy, then formatted a new diskette and copied the image to the new disk. The pair of utilities was written in C; source was available for download from PC Magnet - the *PC Magazine* forum on CompuServe.

I obtained the utilities back when they first came out, and had used them for about a year duplicating small numbers of MS-DOS disks - my MS-DOS business never quite grew to the point of needing a duplicator. However, with an optimistic view of LB86 sales, and the Mountain 3200 available, I proceeded to re-write the 2FLOPPY code to access the Mountain 3200. All I had expected to have to do was to provide the serial port access to control the duplicator and obtain its status. Or so I thought!

The 2FLOPPY code always masked the BIOS error and provided its own error. Thus, whenever I got an error such as *destination disk write error*, I never knew why until I got down into the code and debugged it with CodeView. It was then that I discovered that the errors were the DMA boundary errors. The difficulty in debugging came about because the memory environment changed whenever I ran the program from CodeView relative to running the program from the DOS prompt. As you see now, the error would occur only when the program's data segment origin was such that the allocated floppy I/O buffer crossed a 64K boundary. Believe me, it is difficult to debug a program which always works when you are debugging it but always bombs when you are running it normally. The program's behavior also changed when run on different machines - partly caused by the loading of different TSR modules in a particular machines configuration which resulted in different memory origins for the 2FLOPPY program.

To solve the problem on a permanent basis meant that I had to do one of two things.

1. Write a transfer function which breaks a transfer request into two requests when the memory block is known to traverse a 64K boundary, or

2. Write a memory allocation function which guaranteed to be totally within a segment whose origin was a 64K boundary.

I chose to implement the latter function which I named getbuff().

Memory allocation under Microsoft's C compiler - and probably every other C compiler, allocates requested memory from the heap starting from the bottom and going to higher-numbered addresses. There is no way to force the allocation of a lower-addressed block; however, there is at least one technique which can force the allocation of a higher-addressed block.

Consider the case where unfortunately, the heap (i.e. the data space) starts within 2K of the end of a 64K-boundary block of memory. If I attempt to allocate 4K for an I/O buffer and use it for floppy I/O, I will get a DMA boundary error. So what I want the getbuff() function to do, is to test the memory region obtained for boundary integrity. If it cannot be used for the desired purpose, free it up; allocate a dummy block of memory so that a subsequent allocation will be origined at a higher address. Then test the new block to see if it too cannot be used. Repeat the de-allocation, allocate dummy, allocate buffer sequence until either no more memory is available, or until the desired buffer is found. The former unworkable situation could only occur if a demand for greater than about 32K is requested. That would not occur for the 2FLOPPY program.

One thing which 2FLOPPY does is require a second I/O buffer to read back what was written for comparison with the source which was written. That is an additional level of integrity which virtually guarantees a flawless duplicated diskette. Therefore, 2FLOPPY requires two identically sized buffers.

Enter the C memory function

Lets start pulling apart the getbuff() function to see how it operates.

As mentioned, 2FLOPPY needs two buffers. I therefore specified a global variable *dummy* declared as an array of pointers to char (i.e. char *dummy[2];). What is passed to getbuff() as an argument, is the *number* of the desired buffer. The size of the needed buffer is also a global variable, *buffsize*.

Since getbuff() will be returning a pointer to a memory block, and is passed an integer argument, it is declared as follows:

```
char * getbuff(num)
    int num;
{
```

The function needs five variables: tempsize, an unsigned int used to store the current size of the dummy buffer allocation; buffer, a pointer to char which holds the pointer to the allocated memory block; segment and offset, also unsigned ints used to hold the 16-bit segment and 16-bit offset values; and address, a long integer which will hold the full 20-bit resolved address.

Loops are used in just about every program. C provides three forms of looping: the *for* loop; the *while* loop; and the *do-while* loop. In each case, some expression determines a conclusion to the loop. Sometimes its useful to terminate a never-ending loop by some particular condition which may be more difficult to code into the normal loop termination-testing construct. In such a case, you form a *forever* loop and terminate its looping by a break or return to the calling function. Forever loops can be coded using a *while* construct with an always true condition, such as `while (TRUE) {}`. But since the word *forever* has "for" as a syllable, it is sensible to construct a forever loop using the for construct as in:

```
for (;;)
    {
```

This particular *for* is devoid of initialization, test, and increment statements. The subsequent *for* code block is the entire code of the function. The first statement attempts to obtain a memory block of the needed size. Malloc() returns a NULL if the requested size cannot be allocated; getbuff() just returns this NULL to the calling function to indicate the error condition.

```
  if (!(buffer= (char *)
malloc(buffsize)))
      return buffer;
```

The 20-bit address is calculated in C code with the same method as previously discussed. The segment value is shifted left four bits then added to the offset address. The FP_OFF and FP_SEG are macros defined in Microsoft C and are used to extract the offset and segment addresses given a pointer. Since 2FLOPPY is compiled using a small memory model, the buffer pointer needs to be cast to a far pointer as required by the macros.

```
  address = (offset = FP_OFF(
(char far*) buffer) + (FP_SEG(
(char far*)buffer)<<4));
```

To determine whether the buffer obtained traverses a 64K boundary, I simply strip off the *page* value (the high 4-bits), add the buffer size to the address, then see if the value is less than 65535. If so, the buffer is okay to use for the floppy I/O and I return to the calling function with the buffer pointer.

```
  if (((address & 0xFFFF) +
buffsize) < 0xFFFF)
      return buffer;
```

The alternative is an unusable buffer address. So I free up the buffer, and any previous dummy buffer which may have been allocated.

```
  free(buffer);
  if (dummy[num])
      free(dummy[num]);
```

Since I know that the offset was within a buffer size of the boundary, I simply calculate the difference between the offset origin and the boundary. This tempsize, which when allocated, should guarantee that the next allocation will provide a memory origin at the beginning of a 64K boundary.

```
  tempsize = 0x10000 - offset;
```

With the dummy size calculated, I allocate that sized memory block. The test for a valid allocation guards against the unknown and unconsidered which is bound to happen to every programmer.

```
  if (!(dummy[num] = (char *)
malloc(tempsize)))
      return buffer;
```

With the dummy block allocated, the closing brace of the forever code block is reached, causing a repeat of the first statement of the block.

With the getbuff() modifications made to the 2FLOPPY program, I have not experienced any further DMA boundary errors while running the program on four different machines with quite different runtime memory configurations.

Function code source appears in its entirety on page 18.

Cars, ROMs and 102s

or

A Tandy 102 makes a good speedometer

by James Cameron
Digital Equipment Corporation (Australia) P/L
(cameronjames@snoc01.enet.dec.com)

Here is a good example of how to persuade the Tandy 102 to do two things at once; in this case monitor incoming pulses as well as allow text editing and other programs to run.

Hardware

The hardware is the sensor from a Tandy overspeed alarm, which is connected between two halves of the speedometer cable after cutting it. This provides one pulse for every 0.7 metres of distance in my car. Other hardware options are viable; for instance the magnet and coil approach used in the Electronics Australia January 1991 issue. Whatever the source; pulses must find their way to the Tandy 102's Bar Code Reader port, which is a nine pin "D" socket on the left side. One of the pins of this socket is connected to bit 3 of port X'BB', so by reading from this port, the state of the pin can be determined.

Software

The software consists of two modules, a machine language interrupt routine for taking speed samples, and a BASIC program for the user interface. The rest of this article deals with the software.

Sampling

The sampling routine has gone through three generations so far. These have been;

1. Count up to n pulses and return; the speed is calculated by determining the time taken to count the n pulses. The value n is varied according to the prior speed sample; n is increased if the speed is increased. Problem : no data is returned if the car has stopped, so the display cannot be updated.

2. Count the number of pulses received in n 254ths of a second. Again, n is varied according to the prior speed sample; n is increased if the speed is decreased. Problem : distance measurements are difficult and inaccurate.

3. Maintain an odometer, by providing an interrupt routine that counts pulses and increments the odometer accordingly. The speed is calculated based on the odometer change over time. The odometer is calibrated in pulses, and is 32 bits wide.

I needed to do a lot of reverse-engineering of the 102's ROM, since I didn't have any reference material on entry points or features. What I found of importance was that there is a 254th second interrupt routine vector that I could hook my own program into. A three byte vector high in memory normally containing a RET instruction needed to be replaced with a jump instruction to my own routine.

The sampling program is in two parts; the interrupt driven pulse detector, and the BASIC interface routines. The pulse detector gains control every 254th of a second; it checks to see if the state of the incoming data line has changed from on to off, and if so, increments the odometer value, which is only 16 bits wide at this point.

The interface routines provide a number of functions;

a. start interrupt; installs the replacement vector in high memory,

b. stop interrupt; restores the vector to a RET instruction,

c. clear odometer; resets both the interrupt routine's 16 bit odometer and the internal 32 bit value.

d. return odometer; returns the current value of the odometer to the BASIC program.

e. return odometer at next second; returns the odometer value after waiting for the internal clock to tick over to the next whole second.

User Interface

The BASIC program used for the user interface has also evolved. Initially, it displayed only the speed calculated as a three digit number. Now it displays a horizontal graph with calibration markings, along with a historic vertical graph for the past 240 samples. Function keys allow the sample period to be varied; just for the fun of it. A shorter sample means faster response, but a decrease in accuracy. The program also allows recording of the current odometer setting in a text file for later analysis or trip calculations. Soundeffects are also emitted; with an overspeed alarm setting available.

Calibration

The frequency of the pulses is proportional to the rotation speed of the rear axle or speedo cable. At a given road velocity, the frequency is also dependent on a number of other factors;

a. how well inflated your tyres are,

b. the design of the sensor device, (number of pulses per rotation),

c. the circumference of your tyres,

d. the relative gearing between the tyres and the sensor.

I found (d) to be the most difficult to obtain, since I had no information apart from actual trial and error. I found the ratio in my instance to be 2:5, that is, for every five speedo cable revolutions, two tyre revolutions were observed.

Overall though, I took the accurate approach; count how many pulses are detected in a measured kilometre. Pop out to the Great Western Freeway between Parramatta and Penrith, and watch for the white measured kilometre signs; labelled "mk" on a white background. [US readers are encouraged to do thesame ;-) - JC]

Two listings follow. The first is the machine code for the hardware interface routines, and the second is the working prototype user interface program written in BASIC.

```
;+
;  C3/ASM  (Car device interface, revision 3)
;
; This program runs as an interrupt task monitoring the
data line.
; Each cycle of the data from 0 to 1 and back again
will increment
; a revolutions counter.
;
;-
; symbol prefix abbreviations
; B  byte  W  word  L  longword  P  port  M  bitmask
; R  routine
BASE   EQU 0F5F0H    ; maximum ram address available
;
   ORG BASE-0137H;-K_MODULE
;
P_CABLE      DEFL   0BBH   ; port for speedo cable
M_CABLE      DEFL   8   ; bit within port for cable
R_VECTOR  DEFL   0F5FFH ; interrupt vector
;
FIRST  DEFL   $
   DEC A
   JP  Z,SNAP     ; 1
   DEC A
   JP  Z,SAMPLE   ; 2
   DEC A
   JP  Z,RESET       ; 3
   DEC A
   JP  Z,START       ; 4
   DEC A
   JP  Z,STOP     ; 5
   DEC A
   JP  Z,SPEED       ; 6
   DEC A
   JP  Z,SET      ; 7
   RET
;+
;  Get the time into a buffer, and return a pointer to
the buffer
;-
TIME
   LD  HL,T_NOW
   DI
   CALL   7329H
   EI
   LD  HL,T_NOW
   RET
;+
;  Return an odometer sample; first wait for the time
to tick
;  over to the next second, then grab the sample from
the
;  revolutions counter, then accumulate it into the
odometer.
;
;  Lastly, return both the time and the new odometer
reading
;  in the integer array whose address was passed in HL.
;
;  If A%() is the integer array, then
;
;      a%(0) = lower 16 bits of odometer
;      a%(1) = upper 16 bits of odometer
```

```
;       a%(2) = seconds
;       a%(3) = minutes
;       a%(4) = hours
;-
SNAP
    PUSH   HL
    JP  SEE
;
SAMPLE
    PUSH   HL
;
    CALL   TIME   ; return pointer to
time of day
    LD B,(HL) ; save current seconds
value
WAIT
    PUSH   BC
    CALL   TIME
    POP BC
    LD A,(HL) ; get new seconds counter
    CP B   ; compare
    JP Z,WAIT ; loop until changed
;
SEE
    LD DE,0
    DI
    LD HL,(W_REVOLUTIONS)
    EX DE,HL
    LD  (W_REVOLUTIONS),HL
    EI
;
    LD HL,L_ODOMETER
;
    LD A,E    ; get lsb
    ADD A,(HL)     ; add it in
    LD  (HL),A     ; save it
    INC HL
;
    LD A,D    ; get msb
    ADC A,(HL)     ; add it in
    LD  (HL),A     ; save it
    INC HL
;
    LD A,0    ; zero extend 32bits
    ADC A,(HL)     ; add it in
    LD  (HL),A     ; save it
    INC HL
;
    LD A,0
    ADC A,(HL)
    LD  (HL),A
;
    LD HL,L_ODOMETER; point to odometer
    POP DE    ; point to BASIC array
    LD B,4    ; length of 32bit value
;
LOOP   LD A,(HL)
    LD  (DE),A
    INC HL
    INC DE
    DEC B
    JP NZ,LOOP
;
    LD HL,T_NOW  ; point to clock sample
    LD B,3    ; number of values
;
ROUND  LD C,(HL)     ; get low digit
    INC HL
    LD A,(HL)     ; get high digit
    ADD A,A    ; multiply by ten
    ADD A,A
    ADD A,(HL)     ; x5
    ADD A,A    ; x10
    ADD A,C    ; add in low digit
    LD  (DE),A     ; save in array
    INC DE
    INC DE     ; next 16bit word
    INC HL     ; next value
    DEC B
    JP NZ,ROUND  ; go round and round
until done
;
    RET
;+
;  Start interrupt routine
;-
START
    DI
    LD HL,INTERRUPT
    LD  (R_VECTOR+1),HL
    LD A,0C3H
    LD  (R_VECTOR),A
    EI
    RET
;+
;  Stop interrupt routine
;-
STOP
    LD A,0C9H
    LD  (R_VECTOR),A
    RET
;+
;  RESET
;
;  Reset everything.  HL must be the
revolutions counter required.
;-
RESET
    CALL   SET       ; set the revolu-
tions counter
    DI
    LD  (W_COUNTDOWN),HL
    LD HL,0
    LD  (W_TICKS),HL
    LD HL,0
    LD  (W_REVOLUTIONS),HL
    LD  (L_ODOMETER),HL
    LD  (L_ODOMETER+2),HL
    LD HL,7FFFH      ; minimum speed
sample
    LD  (W_SPEED),HL
    EI
    RET
;+
;  SPEED
;
;  Obtain most recent speed sample.  HL
must point to an array
```

```
;   containing two words into which the
sample is returned.
;   The first word will contain the time
in ticks, the second word
;   will contain the revolution count.
;-
SPEED
    LD DE,W_SPEED
    LD B,4
Z_SPEED   LD A,(DE)
    LD (HL),A
    DEC B
    RET Z
    INC HL
    INC DE
    JP Z_SPEED
;+
;   SET
;
;   Set the revolutions-to-wait-for
counter.
;   HL must contain the value.
;-
SET
    LD (W_REQUEST),HL
    RET
;+
;   Interrupt service routine
;-
INTERRUPT
    PUSH  AF
    PUSH  HL
;
    LD HL,(W_TICKS) ; increment tick
counter
    INC HL
    LD (W_TICKS),HL
;
    LD A,(B_IMAGE)   ; get old image of
port
    LD L,A    ; into l register
    IN A,(P_CABLE)   ; get current image
    AND M_CABLE       ; keep the inter-
esting bit
    XOR L      ; same as previous image
    JP NZ,CHANGED    ; jump if changed
    POP HL     ; restore registers
    POP AF
    RET
;
CHANGED             ; port state
changed
    XOR L      ; get current image again
    LD (B_IMAGE),A  ; save for next
scan
    JP NZ,INACTIVE  ; jump if contact
is open
;
ACTIVE              ; bit has changed
from 0 to 1
    LD HL,(W_REVOLUTIONS)  ; increment
counter
    INC HL
    LD (W_REVOLUTIONS),HL
    LD HL,(W_COUNTDOWN) ; decrement
revolution countdown
    DEC HL
    LD (W_COUNTDOWN),HL
    LD A,H       ; check for zero
    OR L
    JP NZ,INACTIVE      ; jump if not
yet zero
;
    LD HL,(W_TICKS) ; get current
counter
    LD (W_SPEED),HL ; save for caller
    LD HL,(W_SAMPLES)   ; get revolu-
tions to wait for
    LD (W_SPEED+2),HL
    LD HL,(W_REQUEST)   ; get requested
sample rate
    LD (W_SAMPLES),HL   ; set it for
next time
    LD (W_COUNTDOWN),HL ; reset revolu-
tion countdown
    LD HL,0      ; reset tick counter
    LD (W_TICKS),HL
;
INACTIVE
    POP HL
    POP AF
    RET
;
T_NOW     DS 10 ; data from clock chip
W_SPEED       DS 2  ; most recent speed
sample in ticks
          DS 2  ; sample revolutions
W_REQUEST DS 2  ; requested sample
revolutions
W_SAMPLES DS 2  ; revolutions to wait
for
W_COUNTDOWN  DS 2  ; revolution count-
down
W_REVOLUTIONSDS 2  ; revolution
counter
B_IMAGE       DS 1  ; latest port image
L_ODOMETER    DS 4  ; odometer
W_TICKS       DS 2  ; tick counter for
speed sample
;
A0 JP FIRST
;
K_MODULE  EQU $-FIRST
;
ZZZZZZ EQU $  ; must not be greater than
f5f0h
    END
```

The program is divided into a number of sections;

Lines	Purpose
1 to 2	Main Loop
8 to 40	Screen interface subroutine
200 to 910	Function key subroutines
1000 to 1250	Initialisation subroutine
2000 onwards	Development subroutines, Reserve string space and execute initialisation routine.

```
0 CLEAR1024:GOSUB1000
```

Line 1 is the start of the main loop. The program spends most of its time in this loop obtaining speed samples and updating the screen.

Calculate the optimum revolutions to wait for based on the last known speed (K) and the current conversion value (C). Ensure it is within reasonable limits; a zero value corresponds to 65536 revolutions.

```
1 R=K*C:IFR<1THENR=1
ELSEIFR>60THENR=60
```

Make a call to the interface routine requesting a change of revolution count (request code 7).

```
2 CALLU,7,R:
```

Ask the interface for the most recent speed sample (code 6), which is returned in the array T.

```
CALLU,6,VARPTR(T(0)):
```

Calculate the current speed and execute the subroutine that displays it on the screen and updates the graphics.

Note that since the revolution counter change request is deferred by the interface routine, we must calculate the current speed using the revolutions count that was used by the interface. This is returned in the second element of the array. The first element contains the time taken for that number of revolutions in "ticks" of the interrupt clock.

```
K=Z*T(1)/T(0):GOSUB8:
```

Lastly, before doing it all again; check to see if the file logging flag is on, and if so, save this sample into the log file. (Oh; my kingdom for an ENDIF in BASIC!)

```
IFFI%THENPRINT#1,TIME$;K;:
GOTO1ELSE1
```

Line 8 is the start of the screen update routine. It uses the value K (speed in kilometres per hour) and updates the screen. It does a few other things; like make beeping noises when the speed passes a multiple of ten.

First; ignore silly speeds; since they would cause the code to fail with illegal function call errors (value out of range errors).

```
8 IFK>130ORK<0
THENRETURN
```

Next; update the current speed displays if the speed has changed since we last knew it. Update the number;

```
9 IFK0<>KTHEN
PRINT@280,
MID$(STR$(K)+" ",2,3);:
```

then the horizontal bargraph. Here's a chance for some optimisation; in that the code checks to see if it is increasing the bargraph size or decreasing it, and only performs graphics commands on the appropriate part of the bargraph.

```
  IFK>K0THEN
      LINE(K0+25,50)-
(K+24,52),1;BF:K0=K
  ELSE
      LINE(K0+24,50)-
(K+25,52),0,BF:K0=K
```

Now place a new speed sample on the historical vertical bargraph. Put the vertical bar in;

```
20 LINE(X,48)-(X,(48-K/
3)MOD64):
```

increment the horizontal position;

```
X=(X+1)MOD239:
```

and blank out the next bar.

```
LINE(X,48)-(X,0),0:
```

If the speed moves past a multiple of ten, make a noise. If we accelerate from 59 to 61, then we'll here two short tones, the second higher than the first; indicating an acceleration. I've set the tones up with particular intervals, so it's possible to know exactly how fast you are going if you are musically trained.

```
    J1=K\10:
    IFJ1<>J0THEN
        IFJ0>5ORJ1>5THEN
            SOUNDS(J0),5:
SOUNDS(J1),5:J0=J1
        ELSE
            J0=J1
40 RETURN
```

The following sections of code handle the function keys. They are executed automatically when the corresponding key is pressed.

F1 is called "CLEAR". This asks the interface to reset it's odometer settings and any other internal state information.

```
200 'clr
210 CALLU,3,1'reset
215 BEEP
220 RETURN
```

F2 is called "ON". This enables the interrupt portion of the interface routine. To start the program, you would type RUN, then press the F1 then the F2 keys.

```
300 'on
310 CALLU,4'start
315 BEEP
320 RETURN
```

F3 is called "OFF". This disables the interrupt routine. You'd do this if you were leaving the car...

```
400 'off
410 CALLU,5'stop
415 BEEP
```

```
420 RETURN
```

F8 is called "MENU". This just returns to the menu.

Since the interface routine runs as an interrupt, it keeps running. This means that the odometer will keep incrementing even though the operator starts editing a text file or runs something else.

```
500 MENU
```

F4 is called "RECD". This turns on or off recording mode. It also turns on or off a tag on the screen indicating the current state of recording mode.

```
600 'record
610 IFFI%THEN
        FI%=(0>0):
        LINE(239,54)-
(235,50),0,BF
    ELSE
        FI%=(0=0):
        LINE(239,54)-
(235,50),1,BF
620 RETURN
```

F5 is called "MARK". It accepts a location name and records it, the time and the current odometer value in the log file. The main complexity here is the need to avoid the use of the ordinary BASIC INPUT verb, as this will blank the line following entry; which will destroy part of the screen graphics. Instead, the INPUT$() function is used to build our own version of INPUT.

Additionally, the TAB key is used to resample the odometer. This is useful if you would rather enter the description of the location then take an odometer sample. You can even take a sample in the middle of entering the text.

The backspace key is also handled; see line 750.

Variables used here are;

A$	the entered keystroke
B$	the accumulated text line
C$	the time and odometer sample
T()	the odometer sample itself

```
700 'mark
710 B$=""
720 CALL
U,1,VARPTR(T(0)):
    C$= TIME$+
STR$(T(0))+STR$(T(1))
730 PRINT@0,"Mark "C$"
"B$CHR$(27)"K";
740 A$=INPUT$(1):
    IFASC(A$)=9THEN720
750 IFASC(A$)=8THEN
   IFLEN(B$)<>0THEN
   B$=LEFT$(B$,LEN(B$)-
1):
  PRINTCHR$(8)"
"CHR$(8);:
            GOTO740
        ELSE740
760 IFASC(A$)<>13THEN
        B$=B$+A$:
        PRINTA$;:
        GOTO740
770 PRINT#1,"Mark "C$"
"B$
780 RETURN
```

F6 is called "RV down". This halfs the current conversion ratio used to calculate the optimum number of revolutions given the current speed in kilometres per hour. It is this ratio which determines the accuracy of the speed displayed. As the ratio decreases, the accuracy drops in that the difference in displayed speeds becomes greater.

The one advantage to decreasing this ratio is that it enables far more samples to be taken in a given time interval.

```
800 'rvd
810 C=C/
2:SOUND523,1:RETURN
```

F7 is called "RV up". This doubles the conversion ratio; and thus increases the accuracy of the displayed speed at the expense of longer update times. No good in city traffic.

```
900 'rvu
910
C=C*2:SOUND523,1:RETURN
```

The last section; initialisation, is placed here merely to increase the speed of the mainloop, by reducing the number of lines that BASIC has to scan through when resolving a GOTO or GOSUB statement.

I guess this BASIC also finds variables by scanning a list sequentially, so in line 1010 I declare first those variables in order by frequency of use.

```
1000 'initialisation
1010 DEFINT A-Z: DEFSNG
C,Z:
     DIMT(4), K, I, O,
A$, A, B, C, D, L, P1,
P2, R, U, S(12)
1030 U=-2579' address
of interface routine
```

Lines 1050 and 1060 define the ratio used to convert from speedometer cable revolutions per clock tick to kilometres per hour. There are two lines used here because I've been experimenting. The first line is the ratio that I've settled on purely by trail and error; looking at the car's speedometer and comparing against the program's calculated speed. (I wasn't driving at the time).

The second line is what it should be according to theory. 0.7115 is the distance traveled in metres per speedo cable revolution; as measured using chalk and the subroutine at line 2000. 255.7545 is the number of interrupt clock ticks per second; which was measured using the built in realtime clock chip. I can't see the relationship between this value and the published 2.4 MHz clock speed of the Tandy 102. 3.6 is the conversion ratio for converting metres per second to kilometres per hour.

```
1050 Z=775.29862
1060
Z=0.7115*255.7545*3.6'
factor
1070 FI%=(0>0)    '
logging file flag
```

The variable C is the ratio used to calculate the optimim number of revolutions per sample according to the current speed.

```
1080 C=0.6        '
conversion ratio
```

This section sets up the function key names on the label line. The SCREEN verb turns off the label line first, then turns it on when the keys are set up. The last two lines

enable the BASIC function key interrupt logic; whereby the function key routines are called regardless of what the mainline is doing.

```
1090 DATA "Clr", "On",
"Off", "Recd", "Mark",
"Rvd", "Rvu", "Menu"
1100 SCREEN0,0:
     FORJ=1TO8:
         READA$
1110     IFA$<>""THEN
       KEYJ,A$+CHR$(13):
           NEXT
         ELSE
           KEYJ,A$:
           NEXT
1120 SCREEN0,1
1130 ONKEY GOSUB 200,
300, 400, 600, 700,
800, 900, 500
1140 KEYON
```

This section prepares the array that holds the tone values used in the sound effects.

```
1150 FOR J=0 TO 12:
READS(J): NEXT
1160 DATA 0, 0, 0, 0,
0, 12538, 8368, 6642,
5272, 4184, 2092, 1046,
523
1170 OPEN"speeds"FOR
APPEND AS 1
```

This part of the initialisation routine prepares the constant part of the screen display. It is placed here so that it can be invoked separately if required.

```
1200 ' refresh screen
1210 CLS
1220 FOR J=24 TO 154
STEP10: PSET(J,53):
NEXT
1230 FOR J=84 TO 124
STEP20: PSET(J,54):
NEXT
1240 LINE(24,50)-
(24,52)
1250 RETURN
```

Here starts the development and tuning section. The two routines are started by RUN nnnn where nnnn is the line number of the routine.

The first routine is used to display the number of cable rotations detected. It's useful when calibrating the distance trav-

The raw BASIC program follows.

```
0 CLEAR1024,62868: LOADM"speedo": GOSUB1000:
PRINT@0,"o";: FORJ%=0TO3000: NEXT:PRINT@0," ";
1 R=K*C+1:PRINT@0,R;: CALLU,D,R:T=PEEK(I):
POKEVARPTR(T)+1,PEEK(I+1): K=Z*R/T: GOSUB8:
IFFI%THENPRINT#1,TIME$;K,: GOTO1ELSE1
8 A$=INKEY$:IFA$<>""THENGOSUB200
10 IFK0<>KTHENPRINT@280,MID$(STR$(K)+" ", 2,3);: IFK>K0
THEN LINE(K0+25,50)-(K+24,52),1,BF: K0=K ELSE
LINE(K0+24,50)-(K+25,52),0,BF: K0=K
20 LINE(X,48)-(X,(48-K/3)MOD64): X=(X+1)MOD239:
LINE(X,48)-(X,0),0
30 J1=K\10: IFJ1<>J0 THEN IF J0>5 OR J1>5 THEN
SOUNDS(J0),5: SOUNDS(J1),5: J0=J1 ELSE J0=J1
40 RETURN
200 REM interpret keystroke
230 IFA$=" "THENGOTO300
240 IFA$="1"THENGOTO400
250 IFA$="e"THENGOTO500
299 RETURN
300 REM take actual speed sample from user and place in
file
310 PRINT@0,;
320 LINEINPUT"Actual speed ? ";A$
330 PRINT#1,USING"### ### ###";T;R;VAL(A$)
340 PRINT@0,CHR$(27)"K";
350 RETURN
400 RUN"1"
500 MENU
600 'go
610 FI%=(0=0)
620 RETURN
700 'stop
710 FI%=(0>0)
720 RETURN
800 'rv
810 C=C/2:SOUND523,1:RETURN
900 'rv
910 C=C*2:SOUND523,1:RETURN
1000 'initialisation
1010 DEFINTA-Z: DEFSNGC,Z:
DIMT,K,I,O,A$,A,B,C,D,L,P1,P2,R,U,S(12)
1020 I=-2578:REM interrupt counter
1030 U=-2668:REM sample machine routine
1040 D=50' delay to allow settling
1050 Z=775.29862
1060 'Z=0.7115*250.0*3.6' factor
1070 FI%=(0>0)
1080 C=0.6
1090 DATA"Log", "See", "Note", "Go","Stop", "Rv↑",
"Rv↓", "Menu"
1100 SCREEN0,0:FORJ=1TO8:READA$
1110 IFA$<>""THENKEYJ,A$+CHR$(13): NEXT ELSE KEYJ,A$:
NEXT
1120 SCREEN0,1
1130 ONKEYGOSUB400,3000,300,600,700,800,900,500
1140 KEYON
1150 FORJ=0TO12:READS(J):NEXT
1160 DATA
0,0,0,0,0,12538,8368,6642,5272,4184,2092,1046,523
1170 OPEN"speeds"FOR APPEND AS 1
1200 ' refresh screen
1210 CLS
```

elled per rotation. Unfortunately, it won't actually work with the version of the interface routine in use now.

```
2000 GOSUB1000:
     CLS:
     R=1:
     X=0:
     PRINT#1,"sample
dual rotations ..."
2010 CALLU,D,R
2015 IFINKEY$=" "THEN
         PRINT#1,X:
         PRINT@40,X
2020 X=X+2:
     PRINT@0,X;
2030 SOUND600,1
2040 GOTO2010
```

This routine is used to playback a recorded speed sampling session.

```
3000 'see
3010 GOSUB1000:CLOSE1
3020
OPEN"speeds"FORINPUTAS1
3030 T$=INPUT$(9,1):
     B$=""
3040 A$=INPUT$(1,1):
     IFA$<>" "THEN
         B$=B$+A$:
         GOTO3040
     ELSE
         K=VAL(B$):
         GOSUB8
3050 IF NOT EOF(1) THEN
3030
3060 A$=INPUT$(1):
     PRINT@0,;:
     RUN
```

```
1220 FORJ=24TO154STEP10:PSET(J,53):NEXT
1230 FORJ=84TO124STEP20:PSET(J,54):NEXT
1240 LINE(24,50)-(24,52)
1250 RETURN
2000 GOSUB1000:CLS:R=1:X=0:PRINT#1,"sample dual rota-
tions ..."
2010 CALLU,D,R
2015 IFINKEY$=" "THENPRINT#1,X: PRINT@40,X
2020 X=X+2:PRINT@0,X;
2030 SOUND600,1
2040 GOTO2010
3000 'see
3010 GOSUB1200:CLOSE1
3020 OPEN"speeds"FORINPUTAS1
3030 T$=INPUT$(9,1):B$=""
3040 A$=INPUT$(1,1):IFA$<>" "THENB$=B$+A$:
GOTO3040ELSEK=VAL(B$):GOSUB8
3050 IF NOT EOF(1) THEN3030
3060 A$=INPUT$(1):PRINT@0,;:RUN
```

Upgraded Functions for Pro-MC

by J.F.R. Slinkman

Included on the associated DiskNotes 7.3 is an archive which contains a new dfix() function (FDFIX/ASM and FDFIX/REL), a new round() math function (ROUND/C and ROUND/REL), and a new and **much** faster hi-res graphics paint() function (PAINT3/ASM and PAINT/REL).

The new round() function is documented as follows:

round(MATH) **round(MATH)**

This function obtains the rounded value of a double.

```
double round( argx );
double argx;
```

argx - is the double for which the rounded value is desired.

Description:

This function returns the integer closest in value to "argx." Fractional values less than 0.5 will cause "argx" to be rounded down. Otherwise, "argx" will be rounded up.

Warning:

Do not attempt to use this function without a forward declaration, either in the source code or in the MATH.H header file.

Example:

```
#include    <stdio.h>
#include    <math.h>
/*  also #option USERLIB if necessary  */
double round();
char    inbuf[81];
double  d1, d2;
 main() {   puts( "round:  enter your number:  EOF to
exit" );
 while (TRUE)
   {
   if ( !gets( inbuf ) )
      break;
   d2 = round( d1 = atod( inbuf ) );
   printf("d1 = %g, d2 = %g\n", d1, d2 );
   }
}
round:  enter your number:  EOF to exit
-3.33   |d1 = -3.33, d2 = -3
6.666   |d1 = 6.666, d2 = 7
```

Also, when using the new round() function in your programs, you MUST do one of two things: (1) always include a "double round();" forward declaration in your program; or (2), edit your MATH.H header file to add "round()" to the list of extern doubles (on the assumption you'll never use the round() function without also using other double functions requiring inclusion of the MATH.H header).

(NOTE: while you're at it, you should also edit your STDIO.H header by adding the line: "long labs();", as this was omitted in release 1.6, and causes the function to fail if a forward declaration is not included in each and every program that uses it.)

Also included are the listing PAINT3/ASM and the module PAINT/REL. This is a replacement for the old PAINT2/ASM. It's about three times faster, and uses about 70 bytes less memory, than the previous version.

INSTALLATION:

PAINT:

If you have MLIB, load IN/REL, and use the <R>eplace option to replace the old hi-res PAINT module with the new one supplied.

If you do not have MLIB, and already have the hi-res PAINT module in your USERLIB, then you must recreate your USERLIB from scratch, using the new PAINT module instead of the old one.

ROUND:

If you have MLIB, load MATH/REL, and use the <I>nsert option to put ROUND before CEIL.

If you do not have MLIB, add ROUND to your USERLIB.

FDFIX:

If you have BOTH MLIB and SLIB:

The module FDFIX is found in LIBA/

REL. Unfortunately, LIBA/REL is too large to fit in the MLIB buffer. However, if you have the utility SLIB, you can still put the new FDFIX module into LIBA as follows:

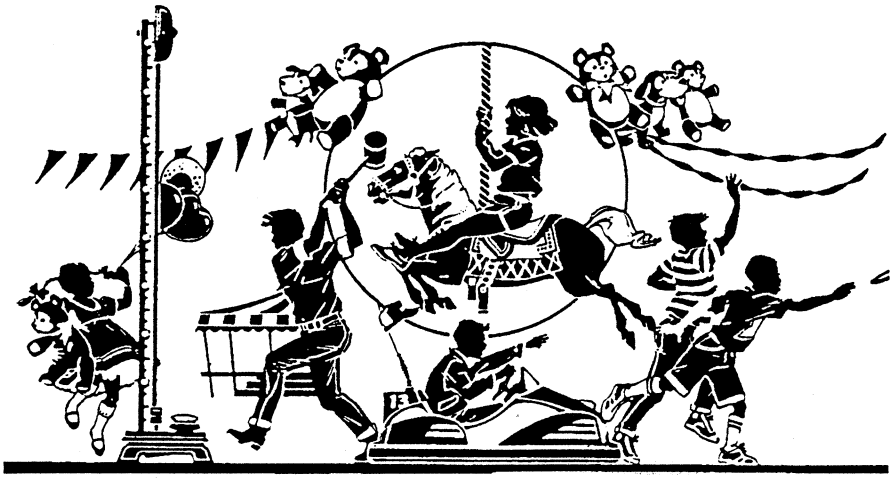
Split LIBA into 2 segments via: SLIB LIBA 9100 :d

Then use MLIB to load LIBA/R01, and use the <R>eplace option, specifying FDFIX. Then <S>ave LIBA/R01 back to disk. Next, rejoin the two segments via: APPEND LIBA/R02 LIBA/R01 (STRIP), followed by a COPY LIBA/R01 LIBA/REL.

NOTE: If you don't have SLIB, you should buy if NOW, as it's cheaper than dirt, and on the MISOSYS, Inc., close-out list.

If you do not have BOTH MLIB and SLIB, you must add FDFIX to your USERLIB.

Have a fun summer

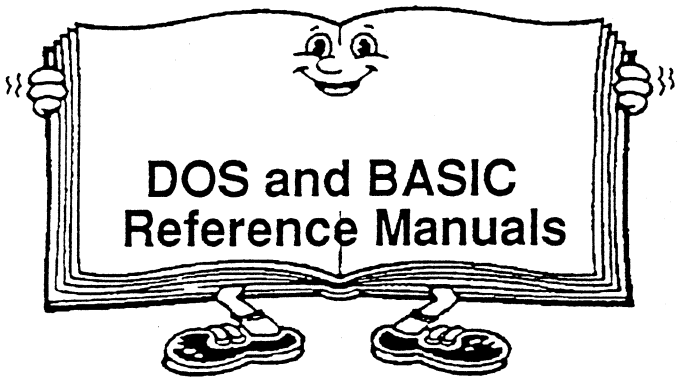


Choose LDOS 5.3.1 or LS-DOS 6.3.1

- ☆ Both Model I and Model III LDOS support similar commands; DOS commands are virtually similar to Model 4 LS-DOS 6.3.1 syntax where possible.
- ☆ The DATE command, "Date?" prompt on boot, and the @DATE SVC now support a date range of 32 years; from **January 1, 1980 through December 31, 2011.**
- ☆ **Enable or disable the printer time-out** and error generation with SYSTEM (PRTIME=ON!OFF).
- ☆ Both ASCII and hexadecimal display output from the LIST command **is paged a screen at a time.** Or run it non-stop under your control.
- ☆ MEMORY displays (or prints) the status of switchable memory banks known to the DOS, as well as a **map of modules** resident in I/O driver system memory and high memory.
- ☆ Specify SYSTEM (DRIVE=d1,SWAP=d2) to **switch drive d1 for d2.** Either may be the system drive, and a Job Control Language file may be active on either of the swapped drives.
- ☆ The TED text editor has commands to **print the entire text buffer**, or the contents of the first block encountered. Obtain directories from TED, too!
- ☆ Have extended memory **known to the DOS?** The SPOOL command now permits the BANK parameter entry to range from 0-30 instead of 0-7.
- ☆ **Alter the logical record length** of a file with "RESET filespec (LRL=n)"
- ☆ Specify "RESET filespec (DATE=OFF)" to restore a file's directory entry to the old-style dating of pre-6.3 release. Specify "RESET filespec (DATE=ON)" to establish a file's directory date as that of the **current system date and time.**
- ☆ SYSTEM command supports removable and reusable BLINK, ALIVE, and UPDATE memory modules.
- ☆ **Double-density BOOT support for Model I** with embedded SOLE and FORMAT (SYSTEM). Supports mirror-image backup, too. Reworked FDUBL driver eliminates PDUBL and RDUBL and takes less memory; enhanced resident driver eliminates TWOSIDE.
- ☆ Model III version auto-detects Model 4 for installation of KI4 keyboard driver; supports CAPS, CTRL, and function keys.
- ☆ SPOOL command offers Pause, Resume, and Clear parameters. (OFF) attempts to reclaim memory used.

MISOSYS, Inc.
P. O. Box 239
Sterling, VA 20167-0239
703-450-4181

- ☆ Both Model I and Model III support similar commands: all features of Model III 5.3.0 are in Model I 5.3.1. That includes such facilities as DOS and BASIC help files, SETCOM and FORMS library commands, TED text editor, BASIC enhancements, etc. All DOS commands have been groomed for Model 4 LS-DOS 6.3.1 syntax where possible.
- ☆ Felt uncomfortable with the *alleged* protection scheme of 6.3? **LS-DOS 6.3.1 has no anti-piracy protection!** Neither does LDOS 5.3.1. MISOSYS trusts its customers to honor our copyrights.
- ☆ Best of all, **a 5.3.1 or a 6.3.1 diskette is available as a replacement for your 5.3.0 or 6.3.0 diskette for $15** (plus $3 S&H in US and Canada, $4 elsewhere). There's no need to return your current master.
- ☆ The 5.3.1 or 6.3.1 diskette(s) come(s) with a 30-day warranty; written customer support is available for 30 days from the purchase date. Versions of 5.3.1 for the Model I and Model III are available. Versions of 6.3.1 for the Model 4 and Model II are available; Model 4 French and German versions are also available (specify 6.3.1 F or 6.3.1 D). Some Model I 5.3.1 features require lower case or DDEN adaptor.



Two new reference manuals are available from MISOSYS. First, we have the the 349-page "LDOS™ & LS-DOS™ Reference Manual", catalog number M-40-060. This single manual fully-documents both LDOS 5.3.1 and LS-DOS 6.3.1 in a convenient 8.5" by 5.5" format. If you use one, or the other, or even both DOS versions, you may want to bring yourself up to date with a single manual. Gone are the many pages of update documentation. Price is $30 plus $5 S&H.

We also publish the "LDOS™ & LS-DOS™ BASIC Reference Manual". This 344-page book, catalog M-40-061, covers the interpreter BASIC which is bundled with LDOS 5.3.1 (even the ROM BASIC portion), the interpreter BASIC which is bundled with LS-DOS 6.3.1, and both Model I/III-mode and Model 4-mode EnhComp compiler BASIC. One convenient 8.5" by 5.5" manual covers all four BASIC implementations for $25 plus $5.00 S&H. Since this new manual covers our compiler BASIC, you can purchase the disk version of EnhComp for $23.98.

MISOSYS, Inc.

With a 20 or 40 MB MISOSYS Hard Drive connected to your TRS-80 Model III or 4, your computer will sail through data access.

MISOSYS has been shipping complete drive kit packages since September 1989 which plug into Model 4/4P/4D and Model III computers; let us build one up for you! Our host adaptor, which interfaces the 50-pin expansion port of the TRS-80 (host) to the 50-pin SCSI port of the HDC, sports a hardware real time clock option using a DS1287 clock module. With its internal battery lifetime in excess of 10 years, never enter date and time again. It even adjusts for daylight saving time! Another option available is a joystick port and Kraft MAZEMASTER joystick with a port interface identical to the old Alpha Products joystick; thus, any software which operated from that joystick will operate from this one.

Software supporting the S1421 and 4010A controllers includes: a low level formatter; an installation utility and driver; a high level formatter; a sub-disk partitioning utility; utilities to archive/restore the hard disk files onto/from floppy diskettes; a utility to park the drive's read/write head; a utility to set or read the hardware clock; a keyboard filter which allows the optional joystick to generate five keycodes; and a utility to change the joystick filter's generated "keystroke" values after installation. Optional LDOS 5.3 software is available.

Twenty megabyte drive packages are currently built with a Seagate ST225 hard drive; Forty megabyte packages use a Seagate ST251-1 28 millisecond drive. Drive packages are offered as '*pre-assembled kits*'. Your 'kit' will be assembled to order and fully tested; all you will need to do is plug it in and install the software. Drive kits include a 50-pin host interface cable and the hardware clock. Full implement of status lights included: power, ready, select, read, and write. Add a joystick or hardware clock for but $20 additional per option (see price schedule).

Aerocomp Hard Drives now available from MISOSYS

MISOSYS is also the sole source of remaining brand new Aerocomp hard drives. All Aerocomp drives include status LEDs, software driver and formatter, power and host cables, and installation Job Control Language. We are building their 20M and 40M drives. We also have Montezuma Micro CP/M Hard Disk Drive drivers available.

Prices currently in effect:

Complete MISOSYS Hard Drive:	
20 Megabyte kit:	$395
40 Megabyte kit:	$495
Joystick option	$20
Hardware Clock Option	$20
LDOS software interface	$30
SCSI software interface	$25
Aerocomp Hard Drives:	
20 Meg unit	$350
40 Meg drive	$450
H/A with MFM software	$75
Separate Hard Disk Controllers	
Xebec 1421 HDC	$45
Adaptec 4010 HDC	$45
WD1002S-SHD	$45
Drive power Y cable	$5
XT drive cable set	$5

Note: freight charges are additional.
Prices subject to change without notice.

Order any hard drive kit or unit from MISOSYS and we'll pre-install either LS-DOS 6.3.1 or LDOS 5.3.1 at no extra charge.

MISOSYS, Inc.
PO Box 239
Sterling, VA 20167-0239
U.S.A.

Contents: Printed Matter

BULK RATE
U. S. POSTAGE
PAID
Sterling, VA
PERMIT NO. 74

Attention Postmaster: Address Correction Requested
Forwarding and return postage guaranteed