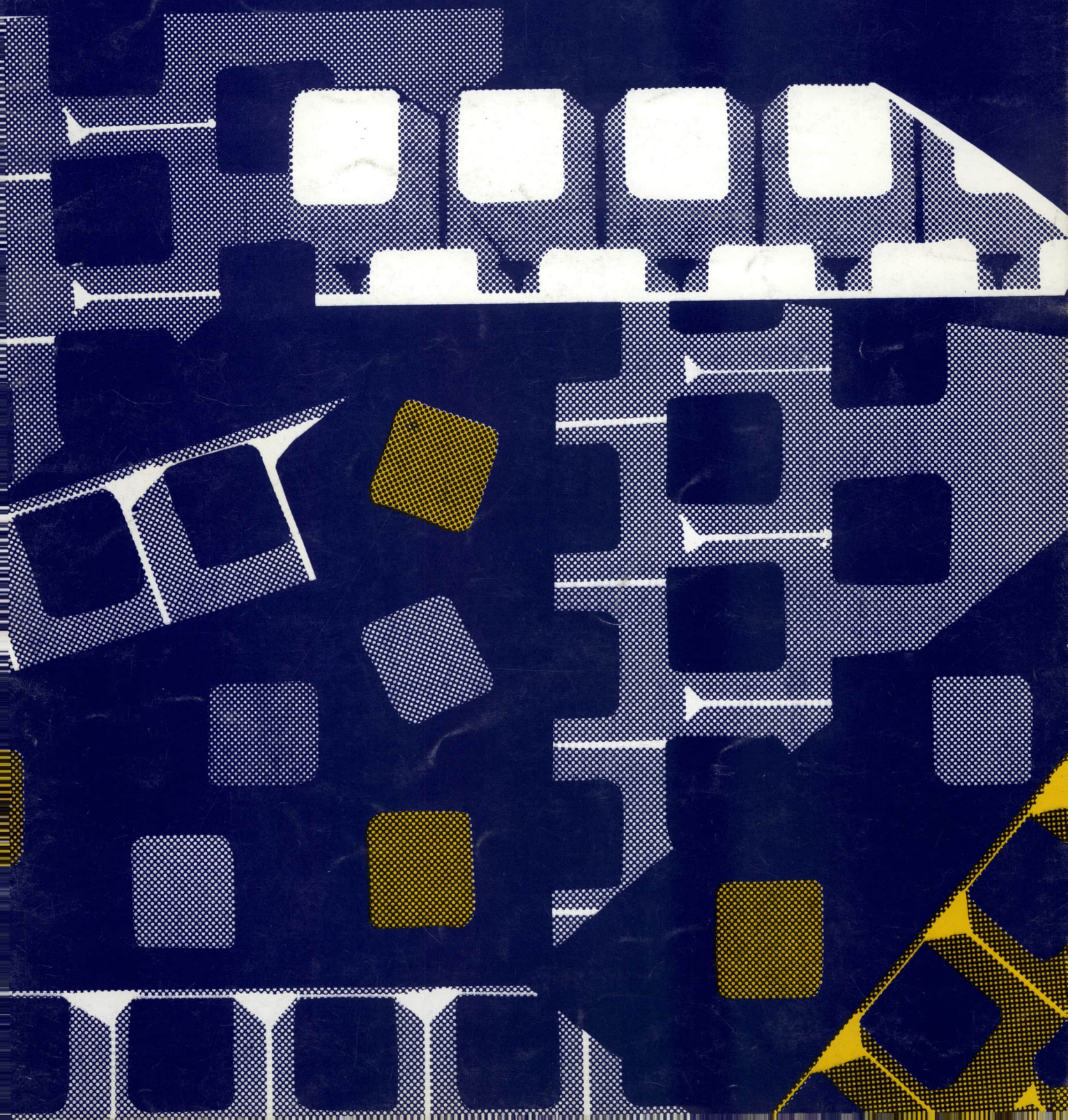
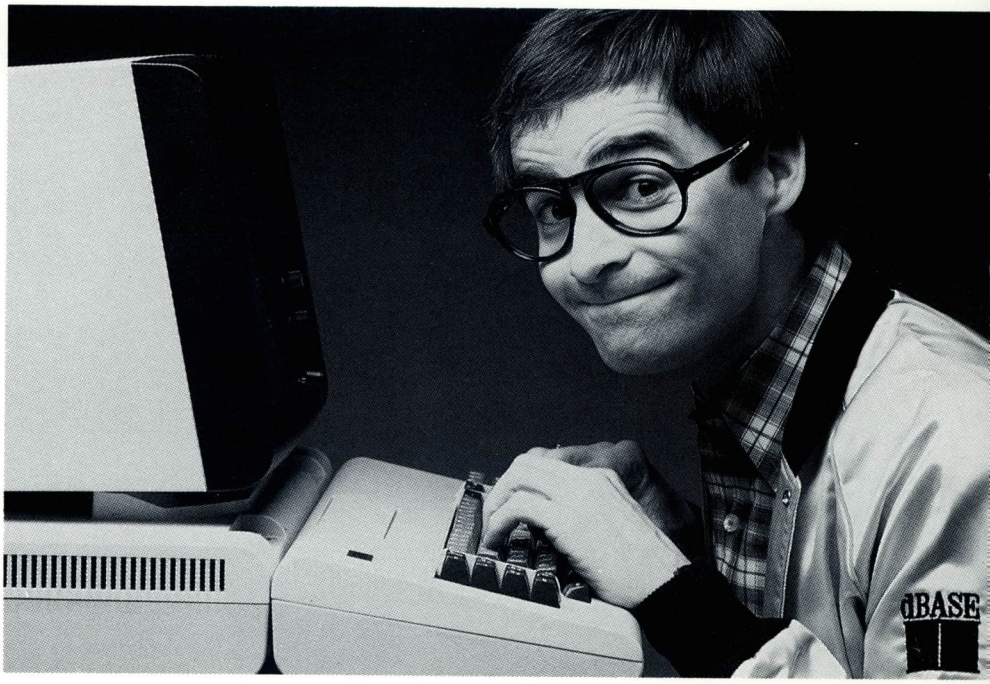
IFELINES™

The Software Magazine™

$3.00 June 1983 Volume IV, No. 1 (SSN 0279-2575,USPS 597-830)



All you dBASE II™ hotshots are about to get what you deserve.



You've written all those slick dBASE II programs.

Business and personal programs. Scientific and educational applications. Packages for just about every conceivable information handling need.

And everybody who sees them loves them because they're so powerful, friendly and easy to use.

But that's just not good enough.

Uh-uh.

Because now you can get the gold and the glory that you really deserve.

Here's how.

We've just released our dBASE II RunTime™ application development module.

And it can turn you into an instant software publisher.

The RunTime module condenses and encodes your source files, protecting your special insights and techniques, so you can sell your code without giving the show away.

RunTime also protects your margins and improves your price position in the marketplace. If your client has dBASE II, all he needs is your encoded application. If not, all you need to install your application is the much less expensive RunTime module.

We'll tell the world.

With your license for the dBASE II RunTime module, we provide labels that identify your program as a dBASE II application, and you get the benefit of all the dBASE II marketing efforts.

We'll also provide additional "how to" information to get you off and running as a software publisher sooner.

And we'll make your products part of our Marketing Referral Service. Besides putting you on our referral hotline, we'll publish your program descriptions and contact information in *dBASE II Applied*, a directory now in computer stores world-wide.

Go for it.

But we can't do any of this until we hear from you.

For details, write RunTime Applications Development, Ashton-Tate, 10150 West Jefferson Boulevard, Culver City, CA 90230.

Or better yet, just call (213) 204-5570. And get what you deserve today.

dBASE™ II

ASHTON·TATE

©Ashton-Tate 1983.

Lifelines

The Software Magazine

Publisher: Edward H. Currie
Editor in Chief: Susan Sawyer
Production Manager: Kate Gartner
Typographers: Paul Blockhaus, Rosalee Feibush
Printing Consultant: Sid Robkoff/E&S Graphics

New Versions Editor: Lee Ramos
Technical Editor: Al Bloch
Advertising Manager: Carolann Abrams
Cover: Kate Gartner

Opinion

2 Editorial — Edward H. Currie

Features

4 High Precision Integer Math Library — Thomas Hill

17 A Programmer's Notes On Quickcode — Marilyn Harper

21 PL/1—You're Going To Love It — Bruce H. Hunter

26 Auto-Start Your CP/M — Steven Fisher

31 A Reveiw of Alpha Software's Data Base Manager — Ron Watson

Software Notes

30 Thunder Clock Routine — David Walker

37 Macro of the Month — Mike Olfe

Product Status Reports

36 New Products

37 New Books

37 New Versions

Copyright © 1983, by Lifelines Publishing Corporation. No portion of this publication may be reproduced without the written permission of the publisher. The single issue price is $3.00 for copies sent to destinations in the U.S., Canada, or Mexico. The single issue price for copies sent to all other countries is $4.30. All checks should be made payable to Lifelines Publishing Corporation. Foreign checks must be in U.S. dollars, drawn on a U.S. bank; checks, money order, VISA, and MasterCard are aceptable. All orders must be pre-paid. Please send all correspondence to the Publisher at the address below.

Lifelines (ISSN 0279-2575,USPS 597-830) is published monthly at a subscription price of $24 for twelve issues, when destined for the U.S. Canada, or Mexico, $50 when destined for any other country. Second-class postage paid at New York, New York, and other locations. POSTMASTER, please send changes of address to Lifelines Publishing Corporation, 1651 Third Avenue, New York, NY 10028.

Program names are generally TMs of their authors or owners. The CP/M User is not affiliated with Digital Research, Inc.
Lifelines—TM Lifelines Publishing Corp.
The Software Magazine—TM Lifelines Publishing Corp.
SB-80, SB-86—TMs Lifeboat Associates
CP/M and CP/M-86 reg. TMs, Access Manager, PLI-80, PLI-86, Pascal MT+, MP/M, TMs of Digital Research Inc.
BASIC-80, MBASIC, Fortran 80—TMs Microsoft, Inc.
KIBITS—TM Bess Garber
Wordmaster & WordStar—TMs MicroPro International Corp.
PMATE—TMs Phoenix Software Associates, Ltd.
Z80—TM Zilog Corp.
Mr. Edit—TM Micro Resource Corp.
MINCE—TM, Mark of the Unicorn

Opinion

by Edward H. Currie

Meanwhile Back on Earth . . .

The only thing that the microcomputer world has spawned faster than the new hardware and software technology is prophets, gurus, soothsayers, experts, geniuses, and visionaries. In the micro industry such terms are used interchangeably! The classic science fiction book "The Dragon's Egg" is in some sense a parable that might well characterize the microcomputer industry.

The story is based upon the premise that an incredibly dense object is hurdling towards earth carrying with it a newly evolved life form which goes through a generation of evolution approximately every 15 milliseconds. As the object approaches earth, manned probes are sent out to investigate. During the course of the subsequent investigations the inhabitants of the incredibly dense object begin to be aware that they are under surveilance. Due to the primitive state of the inhabitants they are initially puzzled. However within a very short period of time they quickly evolve beyond this point to an intellectual state far in advance of the level of sophistication of the earth creatures.

Those of us who are approaching the celestial hemishphere of microcomputerdom are also resting on a base of incredible density but in this case of ignorance. However we, too, are evolving rapidly and the radiation and emanations from this hemisphere are rapidly taking on all the sophistication of a babbling baby. There is sound and fury but most of it signifies little more than the primordial state of the industry and its perennially expounding pontificates. Many of the self-appointed priesthood spin fanciful tales based on virtually little or no understanding of computers and the revolutions they are producing. The industry is for the most part highly incestuous. Its constituents speak loudly and often of "current trends," "the future," and "the role that various newly emerging technologies will play." The industry develops products which in many cases are so complex as to all but preclude their ever reaching a large number of endusers.

They are widely discussed, touted, applauded, and within a short period of time, replaced and/or forgotten.

A litany of allegedly prophetic pronouncements have been dogmatically delivered only which prove to be the prognostications of false prophets.

Among these "orbiter dictums" were:

1) Multiuser operating systems were viable for micros

2) Multitasking was important for micros

3) People would write all their own programs in BASIC

4) Pascal would replace all other languages

5) Program generators would replace generic software

6) Eight bits was dead

7) Computers would get cheaper and cheaper

8) Dealers were the only significant distribution channel for software

10) FORTH was a good general purpose programming language

11) Winchesters weren't viable without some new backup technology other than floppies

12) The 8086 would never get off the ground because of the advent of the 68000

13) The 6800 , 6502, 68000s would replace the 8080, Z80, 8088/8086 processors

14) All software would end up in ROM (to be known as petrified software)

15) That Texas International would wipe out everybody else in the micro market

16) The Apple was a good business machine

17) Parity checking wasn't appropiate for micros

18) Radio Shack would go out of the micro business the same way that everyone left the CB market but sooner.

19) "High Resolution Graphics" was high resolution

20) The IBM-PC wasn't viable without CPM/86

21) Floppy disks would be available for less than one hundred dollars

22) Bubble memory would replace floppys

23) Fifty dollar software would replace all other software

24) Software stores would be as common as record stores

25) Businessmen weren't interested in anything other than accounting software and spreadsheet programs.

26) That micros were suitable for full accounting functions in small businesses

27) Hardware manufactures didn't have to support software purchased by their customers from third parties

28) 2N bits is better than N bits, 256 bits is better than 128 bits, 128 bits better than 64 bits, 64 bits is better than 32 bits, 32 bits is better than 16 bits, 16 bits is better than 8 bits, 8 bits is better than 4 bits, 4 bits is better than 1 bit, 1 bit is better than zero bits.

29) The Japanese would take over the microcomputer software/hardware market the same way they dominated the camera/television/stereo marketplace.

30) Sotware technology could easily keep pace with hardware technology.

31) The Last Program version 2.3 was the last program . . .

32) Good databases for micros were readily available.

33) The most significant use of micros in the home was balancing checkbooks

(continued on page 35)

LIFEBOAT HAS THE ANSWER FOR ALL POPULAR MICROCOMPUTERS *

8-Bit Software Available Today - New Additions Regularly

System Tools
Bug and uBug
Despool
Disilog
Distel
EDIT
EDIT-80
Filetran
IBM/CPM
MAC
MACRO-80
Panel
PASM
PLINK
PLINK II
PMATE
Reclaim
Reformatter
SID
TRS-80 Model II Cust. Disk
Unlock
WordMaster
XASM 05,09,18,.48,51,65 68,75,400, F8, Z8
ZAP80
ZDT
Z80 Development Package
ZSID

Telecommunications
ASCOM
BSTAM
BSTMS-80
RBTE-80

Languages
ALGOL-60
APL/V80
BASIC Compiler
BASIC-80
baZic II
BD Software C Compiler
C BASIC-2
CB-80
CIS COBOL (Standard)
COBOL-80
FORTRAN-80
KBASIC
muLISP/muSTAR
Pascal/M
Pascal/MT + w/SPP
Pascal/BZ
Pascal/Z
PL/1-80
Precision Basic
STIFF UPPER LISP
Timin FORTH
Tiny-C
Tiny C-TWO
UCSD Pascal
Whitesmiths' C Compiler
XYBASIC

Language and Application Tools
BASIC Utility Disk
DataStar
FABS
FABS II
Forms 2 for CIS COBOL
MAG/samE
DES-crypt
dUTIL
MAG/sort
M/SORT for COBOL-80
Programmer's Apprentice
PSORT
QSORT
STRING/80
STRING BIT
SuperSort
ULTRASORT II
VISAM
Access Manager
QUICKCODE

Word Processing Systems and Aids
DocuMate/Plus
Letteright
MagicPrint
MicroSpell
PeachText
SMARTKEY/SMARTPRINT
Spellguard
TEX
Textwriter III
WordIndex
WordStar
WordStar French
WordStar Spanish
WordStar Customization Notes

Data Management Systems
dBASE II
The FORMULA + G.A.S.
HDBS
Hoe
MAG/base 1,2,3
CBS
MDBS
MicroSeed
MicroShell
T.I.M. III
ReportStar
InforStar

Mailing List Systems
CBS Label Option Pak
Mailing Address
MailMerge for WordStar
NAD
Postmaster

Financial Accounting Packages
BOSS Financial/ BOSS Payroll, Accounting System
Financial Pkgs. (PTree)
Financial Pkgs. (SSG)
General Ledger Acctng (Univair)

Numerical Problem-Solving Tools
Analyst
UniCalc
Multiplan
FPL
Microstat
muMATH/muSIMP
PLAN80
Statpak
T/MAKER III
MATH★

Professional And Office Aids
Apartment Management (Cornwall)
Dental Management (Univair)
Dental Management-Family Univair
Friday
GrafTalk
Home Tax
Insurance Agency Management (Univair)
Legal Time Accoutning
Master Tax
Medical Management (Univair)
Medical Management (Family Univair)
PAS 3 Medical
PAS 3 Dental
Professional Time Accounting (PTA)
Property Management Pkg (American Software)
Property Management (PTree)
Tax Planner
Wiremaster

Lifeboat After Hours
Backgammon/Gomoku

Educational Tools
Torricelli Author
Torricelli Studio

Disk Operating Systems
CP/M-80
SB-80

NEW - 16-Bit Software Available for the IBM PC, plus. . .

Conversion Software: 8-Bit/16-Bit
XLT-86
EM-80/86

Emulators
Emulator/86

Word Processing Systems & Aids
MAGIC KEYBOARD
Multi Font
WordStar
MicroSpell
Spellguard

Text Editor
PMATE-86

Data Management Systems
dBASE II
MAG/base1,2,3
T.I.M. III

Numerical Problem-Solving Tools
Math★ PC
PLAN 86
muSIMP/muMATH
UniCalc
Statpak
Multiplan
T/Maker III
FPL

Mailing List Systems
Postmaster
MailMerge

Sorting Tools
PSORT
AutoSort/86M

Professional and Office Aids
The Executive Alert System
Insurance Agency Management
Legal Time Accounting
Tax Planner
Master Tax
Home Tax

Graphic Tools
Halo
FAST GRAPHS

Medical and Dental Aids
Dental Management System
Medical Management System
PAS-3 Dental
PAS-3 Medical

Financial Accounting Packages
The Home Accountant
General Ledger
PeachTree Financial Software

Telecommunications
ASCOM
BSTAM

Networking Tools
PC—NET

Languages
CB-86
CBASIC-86
Lattice C Compiler
muLISP/muSTAR
Pascal MT + 86
PL/1-86
PL/MYCRO-86
PL/MYCRO-87

Language and Applications Tools
FABS PC
MAG/sam-E
WordMaster
DES-crypt
PC-Create

System Tools
The C-Food Smorgasbord
MicroFloat
PL/1-87
UT86
Panel-86
PLIB
PLINK-86
Flight Simulator

Disk Operating Systems
SB-86
CP/M-86
Concurrent CP/M-86
MP/M-86

Apple Software
The Home Accountant
APPLI-CARD
SoftCard
SoftCard Premium System

Hardware and Accessories
Break-Out Box
Diskette Drive Headcleaning Kits
Flippy Disk Kit
Floppy Savers
Smartmodem
Baby Blue
Baby Tex
Quarter Meg
8" Disk Controller
Baby Talk Board

Plus a full line of Books and Periodicals

SEND FOR FULL SOFTWARE DESK REFERENCE WITH DESCRIPTIONS OF ALL THESE PLUS A WHOLE LOT MORE

LIFEBOAT • 1651 Third Ave. • New York, N.Y. 10028

(212) 860-0300 • TWX: 710-581-2524 (LBSOFT NYK) • Telex: 640693 (LBSOFT NYK)

Products are generally TMS or Reg TMS of the Authors

Copyright © 1983 Lifeboat Associates

Feature

High Precision Integer Math Library

by Thomas Hill

Introduction

The program library presented here was adopted from an article published in *Dr. Dobbs Journal* in March of 1977, by M.G. Dineley of Manchester, England. Since first seeing these routines, I have been slowly developing them into a usable library of extended precision math functions. On the whole, they remain in the same form as Mr. Dineley first presented them, but I have spent some time optimizing the code for speed, as will be detailed later on in this article. A large part of the added code deals with applications of the routines to certain number theory problems and to cryptography (a special interest of mine). As far as I can determine, no further works have been published referring to these routines, so I will take it upon myself to be the apparent first to actually present examples of their use, and to provide information concerning time measurements and optimization.

Abstract

There arises in the world of numbers a frequent need to manipulate and perform simple arithmetic operations upon large integer values. Examples of the uses of such a need may be seen in cryptography, natural number theory, the generation of prime numbers, and just plain fun. The set of 8080 assembly language functions described herein have been developed with large integers in mind. They provide the means to manipulate binary representations of large integer values, and include the arithmetic functions of addition, subtraction, multiplication, division, modulus, and square roots. Each function is constructed in a modular fashion, allowing the applications developer to select from a library of available functions.

Possible Uses

The uses to which this package may be put is limited only by the imagination and requirements of the user. I currently have packages which produce large prime numbers, extract the Greatest Common Divisor (GCD) of two integers, produce the Least Common Multiple (LCM) of two integers, generate random numbers, and compute large factorials. Also completed is a package which fulfills a demonstration function, acting as a simple 'desktop calculator" program. Future uses (this was my primary reason for implementing the package) are applications to cryptography, particularly the cryptographic methods described by Martin E. Hellman in the August 1979 issue of *Scientific American*, which use large prime numbers and their products to encode data. If there is sufficient interest, and I have the time, I may prepare further articles describing these packages and their uses.

Implementation

The routines presented here are designed to allow the assembly language programmer to conveniently manipulate large integers. Included are low level sign manipulators, deque utilities, simple positive integer addition and subtraction, generalized addition, subtraction, multiplication, division, modulus, and square root extraction. Each routine is coded as a separate entity, with declared PUBLICs and EXTERNALs for use with relocating assemblers, Microsoft's MACRO-80 in particular.

The integers are represented in RAM as signed binary values, extending from one (1) byte to a maximum of 128 bytes per integer. As may be apparent from the last sentence, the values manipulated do not maintain a constant storage size. Instead, they may expand or contract within the 128 byte maximum, depending upon the precision demands of the routines which manipulate them.

The internal format of integer values is composed of a sign and length byte, and up to 127 bytes of value, stored as a multi-byte binary number. The first byte (byte 0) of the value is the sign and length indicator. The most significant bit (bit 7) is the sign bit: If the bit is one, the value is negative; if zero, the value is positive. The remainder of the byte (7 bits) is a count of the remaining bytes in the value, to a maximum of 127. The special case of zero is indicated by a sign/length byte of zero. The diagram below illustrates the storage format:

[Address]	[Address+1].....	[Address+N]
MSB=sign	Least Sig.	Most Sig.
Rest=length	Byte	Byte

This format allows the manipulation of integer values to 2**1024. Future implementations may expand this to allow integers to 2**32760 by allowing three nibbles for length.

About the Library

Listing 1 is the source code for each of the library modules. Note that they must be separated into individual files before assembly. Each of the library routines is as modular as possible and each contains its own data areas, thus avoiding as much as possible the side effects arising from improper manipulations of common data areas. The development of the library begins (maybe not so obviously) with the very simple routines which retrieve and manipulate the sign and length byte. There are six of these modules, termed LDC1, LDC2, LDB1, LDB2, INRM, and DCRM. These modules are described below:

LDC1 This module loads the C register with the length indicator of the integer pointed to by the HL register pair. The sign bit is stripped.

LDC2 This module operates as LDC1, but will return to the caller's caller if the length of the specified integer is zero.

LDB1 This module performs a function similar to LDC1, loading register B with the length of the integer specified by the contents of the DE register. Again, the sign bit is stripped.

LDB2 This module is analogous to LDC2, operating upon the integer specified by the DE register and returning to the caller's caller on length zero.

INRM This module will increment the length indicator of the integer specified by the HL register. The sign bit is preserved, and the routine aborts to an error message upon overflow.

DCRM This module decrements the length of the integer specified by the HL register. If the resulting length is zero, the integer is set to absolute zero. The ZERO flag is SET on return if the resulting length is zero.

The next group of modules function together to permit manipulation of integer storage space as a double ended queue (deque). These modules use the length indicator utilities previously described as building blocks for integer space manipulation.

PSHL This module will insert the byte in the Accumulator into the low (least significant) end of the multibyte integer pointed to by the HL register. The other bytes are pushed up and the precision (length) is adjusted.

POPL This module removes a byte from the low end of the multibyte data pointed to by the HL register and returns it in the Accumulator. The remaining bytes are pulled back and the length is adjusted accordingly.

PSHH This module inserts the byte in the Accumulator into the high (most significant) end of the multibyte data specified by HL. The length is adjusted accordingly.

POPH This module removes the byte at the high end of the data specified by the HL register and returns it in the Accumulator. The length is adjusted.

Using the preceding modules as a base, the following modules provide low level arithmetic operations, including simple addition, subtraction, left and right rotates, comparisons, and memory to memory moves.

INCR This module increments by one the integer specified by the HL register. The precision is extended if necessary.

LEFT This module will double a multibyte value pointed to by the HL register by a multibyte left shift. Overflow is detected and vectored to the overflow error module. The precision is extended if necessary.

RIGHT This module halves the integer specified by the HL register by a simple multibyte right shift. The precision is adjusted if necessary.

MOOV This module will transfer the multibyte data specified by the DE register to the area pointed to by the address in the HL register. Any data at the destination will be overwritten.

PARE This module compares two multibyte integers. The addresses are passed in the HL and DE registers. Upon return the condition flags are set as follows: ZERO flag is SET if (DE) = (HL), CARRY flag is SET if (DE) > (HL), CARRY is RESET if (DE) < (HL). The two multibyte data strings are left unchanged upon return.

ADD1 This module will add two multibyte data strings together, returning the result in the space pointed to by the DE register. Note that this destroys the previous contents at (DE). This module will add POSITIVE values ONLY.

SUB1 This module will subtract two multibyte data strings, (HL) from (DE). The result is placed at the space pointed to by the DE register. This module will subtract POSITIVE values ONLY.

Now that we have built the foundations, it becomes a simple task to implement the mid-level arithmetic routines outlined below.

AD1 This module provides general addition of both positive and negative multibyte integer values. The arguments are passed as pointers in the HL and DE registers and the result is returned in the data area pointed to by the DE register.

SB1 This module performs the complementary function of subtraction of positive and negative values, subtracting the integer specified by the address passed in the DE register from the integer pointed to by the HL register. The result is returned in the area pointed to by the DE register.

MULT This module provides generalized multiplication of both positive and negative values, returning the result in the area pointed to by the DE register.

DIVM This module provides the Modulo function, returning the remainder of the division: (DE) / (HL). The remainder is returned as an integer, carrying the sign of the dividend. Note that -30 MOD 7 = 2, not 5.

DIV This module is the general purpose divide function, operating upon both positive and negative values. The dividend is passed as a pointer in the DE register, and the divisor is passed in the HL register. The quotient is returned in the space pointed to by the DE register.

DIVR This module operates similar to DIV, except that answers are rounded rather than truncated.

SQRT This module provides the square root of the multibyte integer data pointed to by the DE register. The answer is returned in the area pointed to by the DE register.

SQRTR This module also provides the square root, returning the rounded answer in the area pointed to by the DE register.

This comprises the basic library. Current algorithms for multiplication and division use repeated addition and

(continued on next page)

subtraction, respectively. Future modifications will utilize more efficient algorithms. The balance of the functions included in the library are built up from combinations of the mid-level functions. Two functions of special interest are the input and output modules, HPINPUT and HPOUT respectively.

Listing 2 is the source for the output module, HPOUT. Notice that this is version 2 of the output. The original module used modulo ten operations to recover the individual digits. This version uses a more complex but much faster technique. The benchmark times presented later are for the original output module. Version 2 provides a speed increase of approximately 50 times. HPOUT converts the internal multibyte binary value pointed to by the DE register into ASCII digits suitable for human perusal. Commas are placed at appropriate intervals and, in cases where the integer is longer than one screen line, carriage return, line feed pairs are sent at the first comma location after column 75.

Listing 3 is the source of the input module, HPIN. The input module accepts ASCII input from the console, converting successive digits into the internal multibyte binary format and storing it at the data area pointed to by the DE register. Conversion halts upon encountering a non-digit. Certain characters are trapped and interpreted by the HPIN routine to allow editing of the input. In particular, the Backspace key may be used to delete and back over the last digit(s) entered. The Carriage Return key is trapped and echoed to the console, but is otherwise not acted upon. This allows the entry of integers longer than one screen line without terminating entry.

Timing Considerations

After developing the routines and removing any bugs which crept in, timing tests were initiated. Using a Mountain Hardware clock board in a stopwatch configuration, each of the mid-level modules was timed and least squares formulas developed to describe each module's behavior. Note that each of the times reported below is for the OPERATION only. It does not include the time spent in the supporting program framework. The results may be summarized as follows:

Addition and Subtraction: With the current maximum of 127 bytes per integer, addition and subtraction times will never exceed 1.0 milli-second. This is based upon a sample size of 200 data pairs, composed of 1) the number of digits in the sum (or difference), and 2) the time in milliseconds required for the operation.

Multiplication: Multiplication times depend primarily upon the length of the input arguments. Since the length of the product of two arguments is approximately the sum of the lengths of the arguments, the data pairs for this timing test consisted of 1) the number of digits in the product and 2) the time in milliseconds required for the multiplication operation. The least squares result suggests that the time for a multiplication will never exceed 310 milliseconds.

Division: Timing tests for division (and by extension, for modulo) are particularly sensitive to the lengths of both the divisor and the dividend. After studying the division method and running preliminary timing tests, it was decided to use data pairs composed of 1) the difference in length of the dividend and divisor, and 2) the time in milliseconds. This approach resulted in a least squares equation which suggests that the worst case division (single digit divisor and 300+ digit dividend) would never exceed 2,500 milliseconds. Note that this time rapidly gets better as the difference in length between dividend and divisor decreases in magnitude.

Square Root: The square root algorithm is exceptionally fast, producing a worst case execution time of 2500 milliseconds for the maximum integer value. The data pairs used consisted of 1) the length of the input argument and 2) the time in milliseconds.

Output Conversion: The time spent converting the internal storage format to a form suitable for human comprehension has not been measured to my satisfaction yet, although initial tests seem to indicate that the major bottleneck is here. Initial times predict that conversion of the maximum value may take close to 75 seconds. Note that this figure is dependent upon the transmission rate of the terminal, which in this case is a serial CRT operating at 9600 baud.

Random Number Generation: The random number generator currently is configured to produce values ranging from 1 to 40 digits in length (less than 1/4 the maximum value). Timing tests indicate a new random number can be generated in slightly less than 100 milliseconds.

Assembling the Modules

The modules were assembled with Microsoft's M80 relocating macro assembler. Each module must be assembled separately to produce a .REL file, then the various modules are combined into a .REL library with the Microsoft LIB library manager. The order in which the modules are placed in the library is very important, since the L80 linker makes one pass through any library and all external references must be resolved at that time. The table below shows the module order for the library I am currently using:

Library order, beginning ----> end

PERMUT COMBIN POWER HPOUT2 HPIN NFACT HPLCM HPGCD HPPRIM HPRAND HPX HPDIV HPSQRT HPMULT HPADSB HPSUB1 HPADD1 HPPARE HPMOOV HPRIGH HPLEF HPINCR HPPOPH HPPSHH HPPOPL HPPSHL HPDCRM HPINRM HPLDB2 HPLDB1 HPLD HPLDC1 OVFLW DATA OUTPUT

After the library has been prepared, application programs must be linked to the library using the L80 linker program. Be sure that you specify a data segment and a program segment, or you will have 'time bomb" programs (they blow up at unpredictable moments). Listing 4 is a sample assembly and link session for the upper-level function FACT (factorial). The source listing for the FACT program is shown in listing 5. Listing 6 is a sample run of the factorial test program.

Validation of Mid-Level Modules

Obviously, I am not going to multiply two 70 digit numbers to check the results of the program. If anybody

wishes to do this, more power to him. However, checks were made using the new Hewlett-Packard HP-16C Computer Programmer's calculator, which allows addition, subtraction, multiplication, and division of 64 bit values. Although 64 bits is about 6/100ths of the maximum bit length of the storage format, it was considered that if the results matched at this point, further checking was superfluous. All of the mid-level modules are validated to the 64 bit argument length, and checks were made to lengths longer than this by using the modules "against" themselves. For example: Two numbers were multiplied together, then the product was divided by one of the arguments to return the other argument. This type of test was performed upon all modules to the maximum argument size. All modules passed this test, although it should be noted that the square root routine could not be checked in this manner due to truncation errors. The division routine also showed truncation errors, but they were partially recoverable by using both the division and modulo functions to generate a quotient and a remainder.

Optimization

Once the modules were all written and tested, they were timed using the procedures outlined previously. After developing time estimates for all the mid-level modules, Digital Research's SID program was used in conjunction with the "HIST" histogram utility provided with SID to isolate the "hot spots" in each module. (Hey, that might make a good article: The uses of HIST.UTL and TRACE.UTL in program developments. After isolating the program segments where the most program execution time was occurring, code optimization was performed, replacing certain subroutine calls with in-line code, rearranging program segments, and utilizing Z80 instructions. After optimization, timing tests were again performed. Although appreciable speed increases resulted from program flow manipulations, very little speed increase was seen after Z80 code replacement. If fact, certain routines showed a measurable speed decrease! Some research into the execution times of 8080 versus Z80 instructions seemed to indicate that no significant speed increases would be obtained through the use of Z80 code, so the modules were left in 8080 code for portability.

Further speed increases may be gained by using different algorithms for the multiply and divide routines. In Volume 2 of Knuth's "Art of Computer Programming" an algorithm for multiplication of large binary numbers is described, which appears to allow an increase in speed of some 40 to 50 percent. Other algorithms are presented which may provide even more speed increases, although program complexity will increase, possibly enough to offset the time decreases. Further development is currently being done in this area.

Other areas in which algorithm optimizations may be applied are the input and output conversions (radix changes). Again, Knuth supplies some efficient algorithms which may be implemented at a later date.

Summary

Presented is a set of modules developed in 8080 assembly language and prepared in a modular fashion suitable for use by Microsoft's relocating macro-assembler (M80) and linking loader (L80). Each module is structured from previously defined modules and each module communicates at the data interface, passing arguments as real addresses stored in processor registers. The modules provide the mid-level arithmetic functions of addition, subtraction, multiplication, division, modulo division, square root, input conversion, and output conversion. The defined data format provides integer magnitudes of approximately 300+ ASCII digits length. Validation and timing test results are presented and optimization efforts are discussed. Also discussed is future code optimization by alterations in the algorithms used in the mid-level functions. Later articles will describe in detail the random number generator and the modules designed to produce 1) Factorials, 2) Least Common Multiple, 3) Greatest Common Divisor, 4) Exponential, 5) Prime Number generator, 6) Disk input and output routines, and a simple 8 function calculator developed to demonstrate and test each of these modules in a working environment.

Final Words

There you are. There is probably a couple of months worth of work tied up in these routines. Two of the things on my 'list of things to do' is to extend the storage format to allow more than 127 bytes of mantissa, and to add the framework and bells to produce a reasonable facsimile of an integer BASIC which handles 300+ digit integers. A generalized expression analyzer module which will accept algebraic arithmetic expressions and produce a Reverse Polish stack structure is nearing completion, and should be ready for public consumption by the time this article sees daylight.

I realize that most of you would rather not spend hours slaving over a keyboard entering the source code into your system, so if you want a copy of the source files, plus whatever else I may have developed since you read this, send me $25.00 to cover my copying costs and postage (plus a little to calm my money hungry nature), and I'll send you a single density CP/M disk with source files and documentation. Currently I have completed the following library modules and/or applications:

Prime Number Generator
Greatest Common Divisor
Least Common Multiple
Factorial
Permutations
Combinations
Random Number Generator
Raise to Power
8 Function Calculator
Store Integer to Disk
Generate Public Key Array
(as per M.E. Hellman's article)

I will periodically publish update articles describing new modules and applications (at least until *Lifelines* gets tired of me) which I or others have developed. If you come up with a good application of the library, I would certainly appreciate hearing from you.

References

1. "The Art of Computer Programming," Volume 2, by Donald Knuth. Addison-Wesley, Publishers.

2. "The Mathematics of Public Key Cryptography," by Martin E. Hellman. *Scientific American*, August 1979, pages 146-157.

```
; High precision math functions derived from March
; 1977 Dr. Dobb's Journal originally written by M.G. Dinneley of
; Manchester, England.

; Prepared by Thomas Hill
;              200 Oklahoma St.
;              Anchorage, Ak. 99504

; 08/10/82     Version 1.1
;              Execution speed optimized.

; 06/17/81     Version 1.0
; All operations operate upon integer values stored in
; the following multi-byte format:

;    [Address    ]      [Address+1  ],...      [Address+N]
;     Sign, Length          Least Sig.            Most Sig.
;     Bit 7  Bits 6-0         Byte                  Byte

; Sign bit = 1 indicates negative value.
; Sign bit and length = 0 indicate value of zero.
; Maximum length is 128 bytes, for values of ±1 * 10↑308.

; Algorithms used are:
; Let D be any digit : 0 <= D < R , where R is the Radix
;     N be any arbitrary integer.

; >>>>>>> Square Root:
; Let A be the Argument
;     B be the Working Accumulator
;     C be a Working Accumulator, initially ZERO.

; 1) Choose the greatest B: B = D*R**2*N &
;    B = B*(B+C) <= A
; 2) If B = 0 then GOTO 5
; 3) A = A - B * (B+C)
; 4) C = C + 2 * B : GOTO 2
; 5) Result = C / 2

; >>>>>>>> Divide:
; Let A be the Dividend
;     B be the Divisor
;     C be the answer, initially ZERO.

; 1) Choose the greatest D and the greatest N :
;    D*B*R**N <= A
; 2) A = A - D * B * R ** N
; 3) C = C * R + D
; 4) If N = 0 then GOTO 6
; 5) N = N - 1 : GOTO 1
; 6) Remainder = A : Result = C

; >>>>>>> Multiply:
; Let A be the Multiplicand
;     B be the Product, initially ZERO
;     C be the Multiplier
; 1) D = C Mod R
; 2) C = C / R
; 3) B = B + D * A
; 4) A = A * R
; 5) If C <> 0 then GOTO 1
; 6) Result = B

; ************************************************************

; Description of included routines:

; ** Length indicator utilities:

;    > LDC1 : Uses NIL
;      Loads register C with the contents of the length
;      indicator specified by the HL register — strips the
;      sign bit.

;    > LDC2 : Uses LDC1
;      As LDC1 but returns to caller's caller if result =
;      ZERO.

;    > LDB1 : Uses NIL
;      Loads register B with the length indicator specified
;      by the DE register — strips sign bit.

;    > LDB2 : Uses LDB1
;      As LDB1 but returns to caller's caller if result =
;      ZERO.

;    > INRM : Uses NIL
;      Increments length indicator specified by HL.
;      Preserves sign bit. HALTS on OVERFLOW.

;    > DCRM : Uses NIL
;      Decrements length indicator specified by HL —
;      reduces it to absolute ZERO even if sign bit set.
;      Returns ZERO flag SET if result is = ZERO.

; *** Deque Utilities:
;      These four routines are used to manipulate multi-
;      byte data as a Deque (double-ended queue).

;    > PSHL : Uses LDC1
;      Inserts a byte from the A register into low end of
;      multibyte data specified by the HL register. Pushes
;      up other bytes.

;    > POPL : Uses LDC2
;      Removes a byte from low end of multibyte data
;      specified by HL reg. into A register. Pulls back
;      other bytes.

;    > PSHH : Uses LDC1
;      Inserts a byte from A reg. into high end of
;      multibyte data specified by HL pair.
;    > POPH : Uses LDC2
;      Removes a byte from high end of data specified by
;      HL pair into A register.

; ***** Simple Arithmetic:
;    > INCR : Uses LDC1
;      Increments a value specified by the HL pair and
;      extends precision if necessary.

;    > LEFT : Uses LDC2
;      Doubles a mulibyte value specified by HL pair (Left
;      shift).

;    > RIGHT : Uses LDC2
;      Halves a multibyte value specified by HL pair (Right
;      shift).

;    > MOOV : Uses LDB1
;      Moves a multibyte value from (DE) to (HL).

;    > PARE : Uses LDC1
;      Compares multibyte data from (DE) with (HL). Sets
;      ALL condition flags ZERO if equal, CARRY set if
;      greater than.

;    > ADD1 : Uses LDC1, LDB1
;      Adds multibyte values (HL) to (DE) — result in (DE).
;      POSITIVE values ONLY.

;    > SUB1 : Uses LDC2, LDB1
;      Subtracts multibyte values (HL) from (DE) — result
;      to (DE).
;      POSITIVE VALUES ONLY.

; ***** High level arithmetic:

;    > AD1 & SB1 : Uses ADD1, SUB1
;      Adds and subtracts positive and negative values.
```

```
;    > MULT : Uses LDB2, MOOV, RIGHT, LEFT, ADD1.
;      Work T1 & T2.
;      Multiplies (DE) by (HL) — result in (DE).

;    > DIVM : Uses LDC1, LDB1, MOOV, PSHL, PARE,
;      SUB1, INCR, RIGHT, LEFT. Work T1 & T2.
;      Reduces (DE) modulo (HL).
;      NOTE: –30 MOD 7 = 2 . . . not 5

;    > DIV
;      Divides (DE) by (HL) — Result to (DE)

;    > DIVR
;      As DIV but rounds up on > 0.5 remainders

;    > SQRT: Uses PSHL, ADD1, PARE, SUB1, RIGHT,
;      INCR, MOOV. Work T1 & T2.
;      Square roots (DE) — result to (DE).

;    > SQRTR
;      As SQRT but rounds result.

; ************************************************************

; Routines begin here. Note that they must be separated
; and assembled separately, then combined in a .REL
; library using the Microsoft library manager.

; data areas for use in Divide and Square root functions

        PUBLIC
        T1,T2

        DSEG

T1:     DS      128
T2:     DS      128

        END

=========================================

; Loads register B with length indicator of value at (DE).
; Version LDB2 returns to caller's caller on length zero.

        PUBLIC  LDB1,LDB2
        EXTRN   ABRT

        CSEG

LDB1:   LDAX    D               ; same as LDC1 but use
                                  B and DE
        RAL
        ORA     A
        RAR
        MOV     B,A
        ORA     A
        RET

LDB2:   CALL    LDB1            ; see LDC1
        JZ      ABRT
        RET

        END

=========================================

; Load register C with length indicator of multi-value at
; (HL).
; Version LDC2 returns to caller's caller on length zero.

        PUBLIC  LDC1,LDC2
        EXTRN   ABRT

        CSEG

LDC1:   MOV     A,M             ; get length
        RAL                     : strip sign
        ORA     A               ; clear carry
        RAR
        MOV     C,A             ; load C
        ORA     A               ; check for zero
        RET

LDC2:   CALL    LDC1
        JZ      ABRT            ; zero result
        RET

        END

=========================================

; Abort routine used by LDC2 and LDB2.
; Adjusts stack to return to caller's caller

        PUBLIC  ABRT

ABRT:   INX     SP              ; skip last return
                                  address
        INX     SP              ; return to caller's caller
        RET

        END

=========================================

; High Precision Integer Math overflow routine.
; Currently this routine performs an unconditional return
; to CP/M via a Warm Boot.

        PUBLIC  OVFLW

OVFLW:  LXI     D,OVFLWMSG
        MVI     C,9
        CALL    5
        JMP     0

OVFLWMSG:
        db      0dh,0ah
        db      07, 'Numeric Overflow, cannot complete
        db      calculations.'
        db      0dh,0ah 'Returning to CP/M system
        db      level.',0dh,0ah,'$'

        END

=========================================

; Increment and decrement length indicator of value at
; (HL)
; Sign is preserved. Program vectors to error message on overflow.

        PUBLIC  INRM,DCRM
        EXTRN   OVFLW

        CSEG

INRM:   MOV     A,M
        INR     M
        XRA     M               ; sign change (overflow)
        RP                      ; nope.
        JMP     OVFLW

DCRM:   MOV     A,M
        CPI     81H             ; is it next to zero?
        JZ      LABB
        DCR     M
        RZ                      ; return with zero set for
                                  zero
        XRA     M               ; sign change?
        RP
        JMP     OVFLW
LABB:   MVI     M,O             ; set to absolute zero
```

(continued on next page)

```
        RET
        END
;=======================================
; Dequeue utilities.
; Push and Pop from high or low end of queue formed by multi-byte
; value at (HL).
        PUBLIC  PSHL,POPL,PSHH,POPH
        EXTRN   LDC1,LDC2
        CSEG
PSHL:   PUSHQ   PSW             ; save byte
        MVI     B,0
        MOV     A,M             ; length
        ORA     A
        MOV     C,A             ; to C
        JZ      PSHL2           ; was zero, no need to
                                ; to shuffle data
        DAD     B               ; high end of data
PSHL1:  MOV     A,M
        INX     H               ; shuffle....
        MOV     M,A
        DCX     H               ; ....up....
        DCX     H
        DCR     C               ; ....data
        JNZ     PSHL1
PSHL2:  INR     M               ; advance length
        JZ      OVFLW           ; oops, got too big.
        POP     PSW
        INX     H
        MOV     M,A             ; put new byte away
        DCX     H
        RET
POPL:   MOV     A,M             ; get length
        ORA     A
        RZ                      ; nothing to get.
        MOV     C,A
        PUSH    H
        INX     H
        MOV     B,M             ; get data byte
POPL1:  INX     H
        MOV     A,M             ; shuffle....
        DCX     H               ; ....down....
        MOV     M,A             ; ....data
        INX     H
        DCR     C
        JNZ     POPL1
        POP     H
        DCR     M               ; length = length - 1
        MOV     A,B             ; data removed to Acc.
        RET
PSHH:   PUSH    H
        PUSH    PSW             ; save data byte and
                                  length address
        INR     M               ; make it one longer
        JZ      OVFLW
        MOV     C,M             ; length
        MVI     B,0
        DAD     B               ; to high end of data
        POP     PSW
        MOV     M,A             ; put data in line
        POP     H
        RET
POPH:   MOV     A,M             ; length
        ORA     A
        RZ                      ; nothing to get
        MOV     C,A
        PUSH    H
        MVI     B,0
        DAD     B               ; high end of data
        MOV     A,M             ; get byte at high end
        POP     H
        PUSH    PSW             ; save it
        DCR     M               ; length = length - 1
        POP     PSW
        RET
        END
;=======================================
; simple arithmetic routines
; includes increment value by one, rotate value left one bit ( *2)
; rotate value right (/2)
        PUBLIC  INCR,LEFT,RIGHT
        EXTRN   LDC1,LDC2,INRM,DCRM
        CSEG
INCR:   PUSH    H
        CALL    LDC1            ; get length
        JZ      INCR2
INCR1:  INX     H
        INR     M               ; increment data
        JNZ     INCR3           ; overflow?
        DCR     C
        JNZ     INCR1           ; loop to end
INCR2:  INX     H
        MVI     M,1             ; extend precision
        POP     H
        CALL    INRM            ; add to length
        RET
INCR3:  POP     H
        RET
LEFT:   CALL    LDC2            ; get length
        PUSH    H
LEFT1:  INX     H
        MOV     A,M             ; get byte
        RAL
        MOV     M,A             ; put it back
        DCR     C               ; loop
        JNZ     LEFT1
        JC      INCR2           ; extend precision if
                                ; overflow off end
        POP     H
        RET
RIGHT:  CALL    LDC2
        MVI     B,0
        DAD     B               ; to high end
        MOV     A,M
        RAR
        MOV     M,A             ; rotate right
        MOV     B,A             ; save new top byte
RIGHT1: DCX     H
        DCR     C
        JZ      RIGHT2
        MOV     A,M             ; handle rest of data
        RAR
        MOV     M,A
        JMP     RIGHT1
RIGHT2: DCR     B               ; was new top
                                ; byte = zero?
        RP                      ; no.
        PUSH    PSW             ; yes, save carry
                                ; (= lost bit)
        CALL    DCRM            ; decrement length
        POP     PSW
        RET
        END
```

```
;=======================================

; Move value from (DE) to (HL)

        PUBLIC  MOOV
        EXTRN   LDB1

CSEG

MOOV:   CALL    LDB1            ; length of (DE)
        MOV     M,A
        RZ                      ; was length = zero?
        LDAX    D               ; no, move things
        MOV     M,A             ; store proper length
                                ; count

        PUSH    D
        PUSH    H               ; save pointers
MOOV1:  INX     D
        INX     H
        LDAX    D               ; move....
        MOV     M,A             ; ....byte
        DCR     B
        JNZ     MOOV1
        POP     H
        POP     D
        RET

        END

;=======================================

; Compare two multi-byte values (DE) to (HL).
; Returns ZERO set if equal, CARRY set of (DE) >

        PUBLIC  PARE
        EXTRN   LDC1

        CSEG

PARE:   MOV     A,M
        RAL                     ; sign to carry
        CMC                     ; set to true
        RAR                     ; put it back
        MOV     B,A
        LDAX    D
        RAL                     ; same to (DE)
        CMC
        RAR
        CMP     B               ; compare lengths
        RNZ                     ; lengths not equal
        CALL    LDC1            ; length of (HL)
        PUSH    H
        PUSH    D
        MVI     B,O
        DAD     B               ; point to high byte
                                ; (HL)
        XCHG
PARE1:  LDAX    D
        CMP     M               ; compare bytes
        JNZ     PARE2           ; not equal
        DCX     H
        DCX     D               ; try next byte down
        DCR     C
        JNZ     PARE1
PARE2:  POP     D
        POP     H
        RET

        END

;=======================================

; Simple positive addition and subtraction.
; Addition adds (HL) to (DE), result to (DE).
; Subtraction subtracts (HL) from (DE), result to (DE).

        PUBLIC  ADD1,SUB1
        EXTRN   LDC1,LDC2,LDB1

        CSEG

ADD1:   CALL    LDC2            ; length (HL)
        PUSH    D
        PUSH    H
        CALL    LDB1            ; length (DE)
        SUB     C               ; compare lengths
        XCHG
        JNC     ADD2
        CMA                     ; increase augend
                                ; length to = addend
        ADC     M
        MOV     M,A
        XRA     A               ; clear acc.
        ADD     B               ; length of (DE)
        JZ      ADD5            ; augend = zero, don't
                                ; add
ADD2:   INX     H
        INX     D
        LDAX    D               ; get augend byte
        ADC     M               ; add to addend
        MOV     M,A
        DCR     B
        JZ      ADD6            ; augend exhausted
        DCR     C
        JNZ     ADD2            ; continue addition
ADD3:   INX     H               ; addend exhausted
        MOV     A,M
        ADC     C
        MOV     M,A
        DCR     B               ; continue by adding
                                ; zero to augend
        JNZ     ADD3
        JNC     ADD7            ; finished
ADD4:   INC     H               ; overflow
        MVI     M,1             ; extend data
        POP     D
        POP     H
        CALL    INRM            ; extend precision
        EXCHG
        RET
ADD5:   INX     H
        INX     D
        LDAX    D
        ADC     B
        MOV     M,A
ADD6:   DCR     C               ; add zero to addend
        JNZ     ADD5
        JC      ADD4            ; finished, check for
                                ; overflows
ADD7:   POP     H
        POP     D
        RET
SUB1:   CALL    LDC2
        PUSH    H
        PUSH    D
        CALL    LDB1            ; get lengths
        SUB     C               ; compare
        JNC     SUB2
        XCHG
        CMA
        ADC     M               ; increase minuend
                                ; length (preserve sign)
        MOV     M,A
        XRA     A
        ADD     B
```

(continued on next page)

```
        EXCHG
        JZ      SUB3            ; minuend = 0
SUB2:   INX     H
        INX     D
        LDAX    D
        SBB     M               ; do the subtraction
        STAX    D
        DCR     C
        JZ      SUB7            ; subtrahend exhausted
        DCR     B
        JNZ     SUB2
SUB3:   INX     H               ; minuend exhausted
        INX     D
        MOV     A,B
        SBB     M               ; subtract zeros
        STAX    D
        DCR     C
        JNZ     SUB3
SUB4:   POP     H
        PUSH    H               ; complement any
                                ; negative result
                                ; (2's comp.)
        PUSH    D
        MOV     A,M
        RAL
        CMC                     ; change sign
        RAR
        MOV     M,A
        ANI     7FH
        MOV     C,A             ; length to C
        STC
SUB5:   INX     H
        MOV     A,M
        CMA                     ; complement data
        ADC     B
        MOV     M,A
        DCR     C
        JNZ     SUB5
        POP     D
        JMP     SUB8
SUB6:   INX     D
        LDAX    D
        SBB     C
        STAX    D
SUB7:   DCR     B
        JNZ     SUB6
        JC      SUB4
SUB8:   POP     H
SUB9:   LDAX    D
        CMP     C               ; any reduction in
                                ; precision?
        JNZ     SUB11           ; nope.
        DCX     D
SUB10:  CALL    DCRM            ; reduce precision
        JNZ     SUB9
SUB11:  XCHG
        POP     H
        RET

        END
```

```
==========================================

; general purpose add/subtract routine.
; Add and subtract positive or negative values.
; Adds (HL) to (DE), result to (DE).
; Subtracts (HL) from (DE), result to (DE).

        PUBLIC  AD1,SUB1
        EXTRN   ADD1,SUB1

        CSEG
AD1:    XRA     A               ; clear carry
        JMP     GPAS1
SB1:    STC                     ; set carry
GPAS1:  PUSH    PSW             ; save carry flag
        LDAX    D
        XRA     M               ; differing signs?
        JM      GPAS3
        POP     PSW             ; nope.
        JC      GPAS4           ; carry says subtract
GPAS2:  CALL    ADD1            ; else add.
        RET
GPAS3:  POP     PSW
        JC      GPAS2           ; differing signs, if
                                ; subtract then really add
GPAS4:  CALL    SUB1
        RET

        END
```

```
==========================================

; general purpose multiply
; Multiplies (DE) by (HL), result to (DE).
; Handles both negative and positive values.
; work areas T1 & T2 are declared external data areas

        PUBLIC  MULT
        EXTRN   LDB2,MOOV,RIGHT,LEFT,ADD1
        EXTRN   T1,T2
        CSEG
MULT:   CALL    LDB2
        PUSH    H
        LDAX    D
        XRA     M               ; check signs
        PUSH    PSW             ; save result
        PUSH    D
        PUSH    H
        LXI     H,T1
        CALL    MOOV            ; put multiplicand in T1
        XRA     A
        STAX    D               ; result to zero for start
        POP     D
        LXI     H,T2
        CALL    MOOV            ; multiplier to T2
        POP     D
MULT1:  LX1     H,T2
        MOV     A,M             ; start multiplication
        ORA     A
        JZ      MULT3           ; finished.
        CALL    RIGHT
        LXI     H,T1
        JNC     MULT2           ; least significant bit NOT
                                  set, don't add
        CALL    ADD1
MULT2:  CALL    LEFT            ; left shift multiplicand
        JMP     MULT1
MULT3:  POP     PSW             ; recover signs
        POP     H
        RP                      ; no problem
        LDAX    D
        XRI     80H             ; change sign of answer
        STAX    D
        RET

        END
```

```
==========================================

; general purpose divide, divide & round, and modulo routine
; Divides (DE) by (HL), result in (DE)
; Handles positive and negative values.
; Uses work areas T1 & T2
```

```
        PUBLIC  DIV, DIVM, DIVR
        EXTRN   LDC1,LDB1,MOOV,PSHL,PARE,SUB1
        EXTRN   INCR,RIGHT,LEFT
        EXTRN   T1,T2
        CSEG
DIVM:   XRA     A
        ADI     80H             ; clear carry, set sign
        JMP     DIV1
DIVR:   XRA     A
        STC                     ; set carry, clear sign
        JMP     DIV1
DIV:    XRA     A               ; clear carry, clear sign
DIV1:   PUSH    H
        PUSH    PSW             ; save vector flags
        LDAX    D
        XRA     M               ; sign differences
        RLC
        JNC     DIV2            ; no sign differences
        POP     PSW
        INR     A               ; sign difference in bit 0
        PUSH    PSW
DIV2:   CALL    LDC1            ; length of divisor
        JZ      RETN            ; divisor = 0
        CALL    LDB1
        SUB     C
        JM      RETN            ; divisor > dividend,
                                ; can't divide
        PUSH    D
        INR     A
        MOV     B,A
        MOV     C,A             ; save length difference
        PUSH    B
        LDAX    D
        ANI     7FH             ; clear sign of dividend
        STAX    D
        XCHG
        LXI     H,T1
        MOV     A,M             ; set to T1
        ANI     7FH             ; clear sign of partial
                                ; divisor
        MOV     M,A
        CALL    MOOV            ; move divisor
        XRA     A
        STA     T2              ; zero T2
        POP     D
DIV3:   CALL    PSHL
        DCR     E
        JNZ     DIV3
        MOV     C,A
        MOV     B,D
DIV4:   POP     D               ; division starts
        PUSH    D
        PUSH    B
        LXI     H,T1
        CALL    PARE            ; partial dividend >=
                                ; partial divisor?
        JC      DIV5            ; no
        CALL    SUB1            ; yes — subtract
        LXI     H,T2
        CALL    INCR            ; increment quotient
DIV5:   LXI     H,T1
        CALL    RIGHT           ; right shift partial divisor
        POP     B
        DCR     C               ; loop count
        JP      DIV6
        MVI     C,7
        DCR     B
        JM      DIV7            ; end
DIV6:   PUSH    B
        LXI     H,T2
        CALL    LEFT            ; left shift partial result
        POP     B
        JMP     DIV4            ; continue division
DIV7:   POP     D
        POP     PSW             ; (DE) —> remainder
        JM      DIV11           ; modulo function
        JNC     DIV9            ; no rounding
        PUSH    PSW
        CALL    PARE            ; partial divisor/2 < rem?
        JC      DIV8
        JZ      DIV8            ; no
        LXI     H,T2
        CALL    INCR            ; yes, increment answer
DIV8:   POP     PSW
DIV9:   XCHG
        LXI     D,T2
        ANI     1
        JZ      DIV10           ; no change of sign
        LDAX    D
        ORI     80H             ; change sign
        STAX    D
DIV10:  CALL    MOOV            ; move result to (DE)
        XCHG
DIV11:  XRA     A
        POP     H
        RET

RETN:   POP     PSW
        STC                     ; carry set indicates no
                                ; division
        POP     H
        RET

        END
```

```
;=======================================
; square root routine
; Extracts square root of (DE), result to (DE).
; Uses work areas T1 & T

        PUBLIC  SQRTR,SQRT
        EXTRN   PSHL,ADD1PARE,SUB1,RIGHT,INCR,MOOV
        EXTRN   T1,T2
        CSEG
SQRTR:  ORA     A               ; clear carry for round
        JMP     SQRT1
SQRT:   STC                     ; set carry
SQRT1:  PUSH    PSW
        LDAX    D
        ORA     A
        STC
        JP      SQRT2
        POP     B
        RET                     ; cannot square root
                                ; zero or negative
SQRT2:  MVI     C,1
        LXI     H,T2
        MOV     M,C             ; initialize T2 = 1
        INX     H
        MOV     M,C
        PUSH    D
        MOV     D,A
        MVI     A,0
        DCX     H
SQRT3:  CALL    PSHL            ; make T2 > N (T2 must
                                ; be a square number)
        DCR     D
        JNZ     SQRT3
        LXI     D,T1
        STAX    D               ; clear T1
SQRT4:  CALL    ADD1            ; T1 = T1 + T2
```

(continued on next page)

```
        XCHG
        POP     D
        CALL    PARE            ; N>=T1?
        PUSH    PSW
        CNC     SUB1            ; yes, N=N-T1
        POP     PSW
        PUSH    D
        XCHG
        LXI     H,T2
        PUSH    PSW
        CNC     ADD1            ; yes, T1=T1+T2
        POP     PSW
        CC      SUB1            ; no, T1=T1-T2
        XCHG
        CALL    RIGHT           ; T1=T1/2
        XCHG
        CALL    RIGHT
        CALL    RIGHT           ; T2=T2/4
        MOV     A,M
        ORA     A               ; T2=0?
        JNZ     SQRT4           ; nope.
        POP     H
        POP     PSW
        JC      SQRT5           ; no round, finish up
        CALL    PARE            ; N>T1?
        JNC     SQRT5           ; nope.
        XCHG
        CALL    INCR            ; round result in T1 up.
        XCHG
SQRT5:  CALL    MOOV            ; put result at (DE)
        XCHG
        ORA     A               ; clear carry for good
                                ; result
        RET
        END
```

==========

Alternate Output Conversion for High Precision Math

This is an alternate output radix conversion routine for use with the High Precision math routines. It performs radix conversion using the multi-precision algorithm presented in Knuth's 'Art of Computer Programming,' volume 2, pages 287-288.

In general, the algorithm differs from the present one in the number of steps taken to reach the final decimal output. In the present routine, the internal format integer (binary) is repeatedly divided by 10 and the remainder is output (modulo 10 division). The quotient is repeatedly reduced in this manner until it becomes zero. The effect upon the conversion time is illustrated by the benchmark times for division. Timing tests indicate that division is proportional to the magnitude of the DIFFERENCE in the lengths of the divisor and dividend. This implies that the present method is the slowest that could have been selected.

The alternative method presented here performs the conversion in two steps. The first step is reduction of the input value by a large power of ten, which I shall call BASE1. This produces a series of 'digits' representing the input value in the BASE1. Each 'digit' of this new 'value' is then further reduced modulo 10 to produce the final output string. Note that this method may (and will) produce zero remainders. These must be expanded into strings of zeros equal to the power of ten which BASE1 is set to. I.e.: If BASE1 = 10**12, then a zero remainder from the modulo must be expanded into 12 zeros for the final output.

The advantage of this method is that the time for each modulo reduction is actually reduced due to the decrease in magnitude between the divisor and dividend when compared to the original method. The major disadvantage is the increase in storage space required due to the need for storing the 'digits' of the BASE1 representation and the increased complexity of the program.

Listing 2

```
POWER   EQU     12              ; BASE1 power
MAXDOL  EQU     75

        PUBLIC  HPOUT2
        EXTRN   MOOV,DIVM,T2

HPOUT2: LDAX    .D
        ORA     A               ; check special case of
                                ; zero
        JNZ     STAGE0
        MVI     A,'0'
        JMP     CHROUT          ; print a zero and return
STAGE0: XRA     A
        STA     NEGFLG          ; assume positive
                                ; output
        STA     ANSWER
        STA     CHARCNT
        MVI     A,3
        STA     COMMACNT
        LXI     H,INDEXS
        SHLD    INDEX$PTR       ; set up pointers
        LXI     H,TEMP$ARRAY
        SHLD    TEMP$PTR
        LXI     H,ANSWER
        SHLD    ANS$PTR
        LXI     H,0
        SHLD    OUTCNT          ; and output digit
                                ; counter
        LDAX    D
        ORA     A               ; check sign
        JP      STAGE1
        MVI     A,'-'
        STA     NEGFLG
STAGE1: LXI     H,BASE1
        CALL    DIFM            ; begin conversion to
                                ; BASE1
        JC      S12             ; carry set indicates
                                ; end of conversion
        LHLD    TEMP$PTR        ; pointer to temporary
                                ; space
        CALL    MOOV            ; move the remainder
                                ; to the temporary array
        PUSH    D
        XCHG
        LHLD    INDEX$PTR
        MOV     M,E
        INX     H
        MOV     M,D             ; save address of
                                ; current 'digit'
        INX     H
        SHLD    INDEX$PTR
        XCHG
        POP     D               ; recover remainder
                                ; pointer
        LDAX    D               ; get remainder length
        MOV     C,A
        MVI     B,0
        DAD     B
        INX     H
        SHLD    TEMP$PTR
        SCHG
        LXI     D,T2
        CALL    MOOV            ; put quotient in input
                                ; value
        XCHG
```

```
        JMP     STAGE1          ; continue
; end of conversion, save last remainder
S12:    LHLD    TEMP$PTR        ; pointer to temporary
                                ; space
        CALL    MOOV            ; move the remainder
                                ; to the temporary array
        XCHG
        LHLD    INDEX$PTR
        MOV     M,E
        INX     H
        MOV     M,D             ; save address of
                                ; current 'digit'
        INX     H
        MVI     M,O             ; mark end of index
                                ; pointers
        INX     H
        MVI     M,O
; now have a representation of the input value to BASE1.
; Each 'digit' is pointed to by the addresses in the array
; INDEX. We now begin converting the array values, starting
; with the oldest entry. (Least Significant).
STAGE2: LXI     H,INDEXS
        SHLD    INDEX$PTR       ; reset pointer
S21:    LHLD    INDEX$PTR
        MOV     E,M
        INX     H
        MOV     A,M             ; get a pointer
        ORA     E               ; zero address = done.
        JZ      STAGE3
        MOV     D,M
        INX     H
        SHLD    INDEX$PTR
        LXI     H,TEMP
        CALL    MOOV            ; move it to work area
        LDA     TEMP
        ORA     A               ; zero remainder?
        JZ      EXPAND          ; yep, expand it
S22:    LXI     D,TEMP
        LXI     H,TEN
        LDAX    D
        ORA     A               ; done?
        JZ      S21             ; yep, do another one.
        CALL    DIVM            ; modulo 10
        INX     D
        LDAX    D               ; get the remainder
        DCX     D
        CALL    PUTOUT          ; save it
        XCHG
        LXI     D,T2
        CALL    MOOV            ; put the quotient back
        XCHG
        JMP     S22
EXPAND: MVI     B,POWER         ; power of BASE1
EXP1:   MVI     A,O
        CALL    PUTOUT          ; send zeros
        DCR     B               ; for POWER times
        JNZ     EXP1            ; for POWER times
        JMP     S21
; have now converted input into decimal valued digits.
; Now output them, placing commas every third digit.
STAGE3: LDA     NEGFLG
        ORA     A               ; check sign
        JZ      S30
        CALL    CHROUT          ; print negative
S30:    LHLD    OUTCNT
        XCHG                    ; output count in DE
        LXI     H,ANSWER
        DAD     D               ; start at the other end
        DCX     H
S34:    MOV     A,M
        CPI     0
        JNZ     S33
S35:    DCX     H
        DCX     D               ; skip leading zeros
        JMP     S34
S33:    CPI     ';'-30H
        JZ      S35             ; also skip leading
                                ; comma
S31:    MOV     A,M
        ADI     'O'             ; make ASCII
        CALL    CHROUT
        LDA     CHARCNT
        INR     A
        STA     CHARCNT
        MOV     A,M
        CPI     ';'-30H         ; was comma sent?
        JNZ     S32
        LDA     CHARCNT
        CPI     MAXCOL          ; near end of line?
        JC      S32             ; nope.
        CALL    CRLF            ; yep, start new line
        MVI     A,O
        STA     CHARCNT
S32:    DCX     H
        DCX     D
        MOV     A,E
        ORA     D
        JNZ     S31             ; loop on digit count
CRLF:   MVI     A,ODH
        CALL    CHROUT
        MVI     A,OAH           ; follow with CR,LF
        CALL    CHROUT
        RET
CHROUT: PUSH    H
        PUSH    D
        PUSH    B
        MOV     E,A
        MVI     C,02
        CALL    5
        POP     B
        POP     D
        POP     H
        END
PUTOUT: PUSH    H
        PUSH    D
        PUSH    B
        PUSH    PSW
        LHLD    ANS$PTR         ; answer buffer pointer
        MOV     M,A
        INX     H
        LDA     COMMACNT
        DCR     A               ; put commas in place
        STA     COMMACNT
        JNZ     PUT1
        MVI     A,';'-30H
        MOV     M,A             ; put a comma in
        INX     H
        MVI     A,3
        STA     COMMACNT        ; new comma count
        XCHG
        LHLD    OUTCNT          ; count the comma
        INX     H
        SHLD    OUTCNT
        EXCHG
PUT1:   SHLD    ANS$PTR
        LHLD    OUTCNT
        INX     H               ; count the digit
        SHLD    OUTCNT
        POP     PSW
        POP     B
        POP     D
```

(continued on next page)

```
            POP     H
            RET

            DSEG
; data
NEGFLG:     DB      0
OUTCNT:     DW      0
CHARCNT:    DB      0
COMMACNT:
            DB      3
TEN:        DB      1,10
BASE1:      DB      5,00,10H,0A5H,      ; 10↑12
INDEX$PTR:
            DW      INDEXS
TEMP$PTR:
            DW      TEMP$ARRAY
ANS$PTR:    DW      ANSWER
TEMP:       DS      128
INDEXS:     DS      128                 ; room for 64 entries
TEMP$ARRAY:
            DS      512                 ; guess
ANSWER:     DS      255                 ; should be enough
            END
```

; This subroutine will accept decimal digits for use as input *(Listing 3*
; values for the high precision arithmetic routines.
;
; The multi-byte value buffer for input is assumed to be pointed
; to be the DE register.
;
;

```
            PUBLIC  HPINPUT,INCNT
            EXTRN   MULT,AD1,DIV,OUTPUT
; equates
            CPM     EQU     0
            BDOS    EQU     CPM+5
            CONIN   EQU     1
            CONOUT  EQU     2
            BS      EQU     08
            CR      EQU     0DH
            LF      EQU     0AH

HPINPUT:    XRA     A
            STA     NEGFLG
            STA     INCNT               ; zero counters
            STAX    D                   ; set value to zero
                                        ; initially
GETNUM:     PUSH    D                   ; save buffer pointer
GETNM1:     MVI     C,CONIN
            CALL    BDOS                ; get a digit
            CPI     03                  ; return to CP/M on
                                        ; control-c
            JZ      CPM
            CPI     BS                  ; delete last digit if
                                        ; backspace
            JZ      DELETE
            CPI     CR                  ; goto new line without
                                        ; disturbing value
            JZ      NEWLN
            CPI     '-'                 ; if minus, check for
                                        ; valid negative input
            JZ      NEGVAL
            CPI     '0'                 ; now check for valid
                                        ; decimal digit
            JC      DONE
            CPI     '9'+1
            JNC     DONE
            STA     DSAVE               ; save the digit
            POP     D                   ; recover pointer
            LXI     H,TEN
            CALL    MULT                ; multiply by ten
            LDA     DSAVE               ; recover digit
            SUI     30H                 ; remove ASCII bias
            ORA     A
            JZ      GETNUM              ; if zero no need to
                                        ; add in
            STA     DIGIT+1
            LXI     H,DIGIT
            CALL    AD1                 ; add to new digit
            JMP     GETNUM
DELETE:     POP     D                   ; recover pointer
            LDA     INCNT
            ORA     A                   ; if digit count is zero,
                                        ; don't do anything
            JZ      GETNUM
            LXI     H,TEN
            CALL    DIV                 ; remove last digit from
                                        ; value
            PUSH    D
            MVI     A,' '
            CALL    OUTPUT
            MVI     A,BS
            CALL    OUTPUT
            LDA     INCNT
            DCR     A                   ; decrement digit count
            STA     INCNT
            JMP     GETNUM
NEWLN:      MVI     A,CR
            CALL    OUTPUT
            MVI     A,LF
            CALL    OUTPUT              ; print a cr,lf
            JMP     GETNM1
NEGVAL:     MOV     B,A                 ; save character
            LDA     INCNT               ; if digit count is zero,
            ORA     A                   ; then accept value as
                                        ; negative.
            JNZ     DONE1               ; else assume digit
                                        ; terminates
            INR     A
            STA     NEGFLG              ; set negative flag
            JMP     GETNM1
DONE1:      MOV     A,B
DONE:       POP     D
            PUSH    PSW                 ; save character
            LDA     NEGFLG
            ORA     A                   ; is value negative?
            JZ      DONE2               ; nope.
            LDAX    D
            ORI     80H                 ; set sign negative
            STAX    D
DONE2:      POP     PSW
            RET
; data areas
            DSEG
TEN:        DB      1,10
DIGIT:      DB      1,0
NEGFLG:     DB      0
INCNT:      DB      0
DSAVE:      DB      0
            END
```

Link of program TSTFAC using Microsoft's L80 linker: *(Listing 4*

```
AO>L80<cr>                  <— Invoke link program at CP/M
                               command level

Link-80     3.37    08-May-80   Copyright 1979,80 ©Microsoft

*/P:100,/D:1000             <— Set program at 0100H,
                               data at 1000H

*B:TSTFAC                   <— Tell L80 what program to link
Data        1000    10CA
Program     0100    0130    <— L80 tells us where things went
                               and how big
```

continued on page 29

Programmer's Notes On Quickcode

by Marilyn Harper

This report is based on 4 or 5 months of using Ashton-Tate's dBASE II (version 2.23B) and Fox & Geller's Quick code application generator (version 2.0A), on a Radio Shack Model II. I have read a number of tips and notes on dBASE II, but little about Quickcode. These are some of the methods I have discovered to help me get the best performance out of this valuable programmer's tool. Also included are notes on some of the many invalid things I tried to do, which should be avoided.

Customizing the Quickcode controls with the C (Configure) Menu. The ability to configure the screen commands to your own preferences is one of the nice features of QC. Although QC's program generation does about 75% of the work for me, I do a lot of editing of the results with MicroPro's WordMaster. I reconfigured QC to the nearest WM equivalent commands, and found this improved the ease of use of QC tremendously, since I don't have to constantly switch between two wildly different command sets. When you buy QC, configure it to match your favorite editor as closely as possible. Then your fingers will fly in Quickscreen mode.

Using Text Files with Quickcode. My interest in using ASCII text files with Quickcode stems from the inability of QC Version 2.0A to insert blank lines into existing screen image files. In designing complex data entry screens, one sometimes finds it necessary to add data fields to existing screens. (This is especially true when the end user isn't really sure at first what he/she wants.) Quickcode 2.0A can delete lines from its screen image file, but there's no way to push existing lines down. Of course, you can add new fields below the last old line you used, but this may not be what you want to do.

It occurred to me to try to read a QC .SCR file, renamed to .ZIP, into the Aston-Tate ZIP screen generator, which can insert blank lines. But alas, the .SCR file is in a non-ASCII format which can't be read by either ZIP or WordMaster.

Finally I found the solution to my problem. As long as I am putting together a simple screen, I use QC as usual. But for a complicated screen or one I think the user may want to have modified, I use the following procedure:

I type in the first-draft screen under Wordmaster, as SCRNAME.TXT. This file can be loaded into QC as a text file. The only insoluble problem I have encountered with a text file source is that tab characters inserted into the ASCII file are not processed by QC to give the result one would expect. It's simpler to avoid the Tab key while in your editor. Also, you may need to start your WM file 1 or 2 lines down, depending on whether you use the Automatic Pilot feature of QC. (I don't.)

Parenthetically, the S menu of QC allows you to turn off the Pilot, turn off the status line, and set screen length to 24 lines. By making these adjustments, you could use all 24 lines in Quickscreen mode. Thus a text file may be up to 24 lines long, as stated in the QC manual. Under normal circumstances, you probably won't want to use all 24 lines. For one thing, dBASE version 2.23B now preempts line 0. I tried to 'cheat' on a crowded screen by adding my screen-title string on line 0 with WordMaster, but when I ran the program, dBASE cleared a piece of that string when the screen first came up. A section of line 0 is used for runtime messages like INSERT ON during reading of @ GETs. Short titles can still be squeezed onto line 0 provided you use the edges of the screen.

Personally, I like the QC Line/Column monitor on the status line, and never turn it off. It's especially handy when I've predetermined a screen layout on paper. It makes it very easy to be sure I type Quickscreen fields to match the paper layout.

Back to text files: with QC using line 0, text entered on line 1 of a text file will be overlaid by the status line. This means you won't see it in Quickscreen mode, except briefly when you first enter this screen mode. However, any characters on the top line of a WM file show up in an @ SAY on line 0 in the programs QC writes for you. This is true even if the S menu says the top margin is screen line 1!

QC behaves strangely with text files below screen line 20 unless you explicitly set the bottom margin lower than this. You'll be able to see any characters on lines 21-23, but you can't move the cursor there.

Using QC with a text file, I now have a quick-and-dirty applications screen which the user can begin to try out. If I am later asked to modify the screen, it is very easy to do so. If I need to add fields to the screen, or just juggle the existing fields into a new format, I can use the full editing capabilities of WordMaster in my SCRNAME.TXT file. If for some reason I want to keep the original screen image and programs, named in this example SCRNAME.TYP, I can simply RENname or PIP a new-named copy of the ASCII screen image before editing and loading it into QC for generation of a new set of programs.

The ASCII file I create under WM can also be used as program documentation, similar to QC's .PRN file. The .PRN file prints as hardcopy showing your labels and has colons to mark the data fields. Printing my file .TXT shows the labels and the variable name assigned to each field. The end-of-field markers (usually >) will be printed if you have typed them into your text file.

Incidentally, you can get some bizarre effects if you don't follow the QC manual's instructions for text files (brief though these are). You're supposed to select T from the main menu, and then get a prompt for the filename. If the text file is

(continued on next page)

NAME.TXT, you should only enter NAME in response to the prompt. But let's say that you select the T main menu option, and then forget and type in the full filename, NAME.TXT, when prompted. QC will cheerfully load your ASCII file under the full name. You now have a major problem which you won't know about until you key for program generation.

Quickcode makes all sorts of convincing noises on the disk while it tells you it has SUCCESSFULLY CREATED NAME.TXT.IO, NAME.TXT.FAU, etc. Of course, you won't find any such files in the directory if you look for them later. You will also find that your original NAME.TXT file has been zapped by something that QC has written over it. This new file appears to be 'NAME.TXT.PRN,' but it unfortunately shows up on the disk under the name NAME.TXT.

Another thing to watch out for is forgetting that NAME.TXT is an ASCII file. If you try to load it under the menu selection O (OLD screen), which QC reserves for its own .SCR file format, you'll probably bomb out of QC with a fatal file error. Some or all of QC is obviously written in Digital Reseach's PL/I-80, so if you've ever used that language, you'll probably recognize some of the error messages. When I tried to load files generated by ZIP into QC as OLD screens, I managed to achieve error conditions so drastic they required cycling the power to get the system unlocked.

Tricking Quickcode into doing the screen My Way. I want my data screens to look good. I get a lot of 'looks' for free with dBASE on the TRS-80 Model II, because with INTENSITY on, all of the @ SAY's are presented in reverse video. This makes a nice, no-effort-on-my-part separation between the labels, screen titles, etc., and the @ GET data fields.

Quickcode doesn't produce precisely the screen I want without a few tricks on my part. I want every character string which will appear in reverse video to have an extra blank RV character at the beginning and end of the string. This makes the edge characters, especially if they're upper-case letters, much easier to read. I also want to avoid unpleasant stair-case effects, which are hard on the eyes when they're highlighted in reverse video. Although I always lined up the data fields vertically, my earliest QC screens looked like patch-work quilts of light and darkness.

Here is an example of how I would enter several lines in Quick screen mode (or with Wordmaster) to get QC to generate the programs my way:

```
Q** Screen Title **Q

QLong long long labelQ  ;VAR:NUM1 > QNext label Q ;VAR:NUM2 >
QQQQQQQQShorter labelQ ;VAR:NUM3 > QQQLabelQ ;VAR:NUM4 >
QQQQQQQQQQQQQLabelQ ;VAR:NUM5 > QThis labelQ ;VAR:NUM6 >
```

All those Q's are incorporated into the label strings of the programs which QC writes for me. I can then load SCRNAME.IO, for example, into WordMaster, and make a global substitution of all Q's to blanks without changing the position of the string delimiters. (I chose 'Q' because it rarely appears in the words of my labels, but if it did, of course I'd edit that word back to the correct spelling.) When the dBASE program is executed, the six labels above will appear as two highlighted, rectangular blocks. There are no rough, irregular edges at all, and QC did practically all of the work.

Parenthetically, if you want to use the apostrophe in a string, you may find that dBASE will only SAY the string up to that point. If the line of code as produced by QC is:

@ 10,00 SAY ' Customer's Name '
just edit this to:

@ 10,00 SAY " Customer's Name "
to get the desired string to print.

Dealing with long fields, labels, and titles. QC won't allow you to set up a character field longer than 79 spaces. dBASE itself will accept up to 254. If you need a character field longer than 79, set up the field in Quickscreen and leave one or two lines immediately below it blank. Let QC assign its maximum length to the field. After program generation, you can edit the .FAU program, as described below, to achieve the longer length you want.

I never bother to put default values in character variables via fields mode, even when they're shorter than the allowed 11 spaces. I edit them into the .FAU program with WordMaster. The .FAU program consists of string assignments to memory variables, as in:

STORE ' ' TO MDLR:ADDR

It's a snap to type my default string into the space between the single quotes.

To lengthen a string beyond 79 spaces, just move the cursor inside the quotes, turn insert on, and count strikes on the space bar up to the length you want. When you run the screen program, dBASE glibly 'wraps' the long field down to the next line, as needed, and places the end-of-field marker in the right place. If you use long labels or titles, QC chops them up into 20-character strings (or less). This can create a messy-looking program. I use WM to edit these fragments back together. The program listing not only looks better, but is shorter and more readable.

Dealing with short data fields. If you ever have to build a QC screen with data fields which are shorter than the variable name you planned to assign to the field, you may be in for some surprises. Let me illustrate with an example.

Say you're in Quickscreen mode and you type the following:

Label1 ;VAR1>
Label2 ;VAR2>

but you want the VAR1 field to be only 1 character in size. You realize you can't put the > end-of-field marker directly where it should be, so perhaps you go into fields mode and change the LEN column from 5 to 1.

QC will tell you the screen has been adjusted. But now when you go back to screen mode, you will see that QC has done what you intuitively avoided doing:

Label1 ;>VAR1
Label2 ;MVAR2>

If you freak out and immediately try to return to fields mode, your system will probably lock up and require a cold restart. QC has in effect deleted the variable name for the MVAR1 field. This condition is fatal if you

enter fields mode. Similarly, entering fields mode with a semicolon anywhere on the screen followed by a blank space, i.e. no valid field name, locks up the Model II, at any rate, faster than your sister.

The trick with short fields is to avoid using the end-of-field marker at all. Type in the full field name you want, and let QC assign the default field width of the remaining space to the end of that line. Then you can safely go into fields mode and change the field width from 70 or whatever to 1. This 'seals off' a short field without altering what you see on the screen in Quickscreen mode.

When placing several short fields on the same line, I've found it to be a good idea to 'seal off' the left-most field this way, before I type in the label of the next field. I've had some problems of the following sort:

Label1;M1 Label2;M2 > Label3;M3>

If you type the line exactly as it appears above, and then go to fields mode, QC sees only two fields. The first is named 'M1 Label', and is 18 chars in length, but is flagged with an 'illegal char' code in the ER column. The second is M3, which is 33 chars in length. This problem arises only when you want to use a very short field, followed hard by the label of the next field. You can't close off M1 with the '>' if the field is destined to be shorter than its variable name. But in the case above, 'Label2' falls within the 9-character space QC allots to the first field. None of these difficulties arise if you simply type in Label1 ;M1 and 'seal off' this short field in fields mode, then return to Quickscreen mode to type in Label2.

Using Integer Data. A nice feature of QC is the 'extra' data types such as Telephone and Date. These are handled automatically by generation of PICTUREs:

```
@ 05,10 SAY "Today's Date " GET
  MDATE PICTURE '99/99/99'
```

Oddly, QC produces no PICTURE for an integer field. This results in all integer GETs having an 11-digit field presented on the run-time screen. This is irritating when you have taken the trouble with QC to set field INT1# to a length of, say, 6. Doubly irritating is having the run-time field actually accept up to 11 digits, and then inform you in a wordy error message at the bottom of the screen that the input is too long.

The only sensible remedy with this version of QC is to edit the PICTURE into the GET statement yourself. Find your integer GETS in .IO and change:

```
@ 02,30 GET MINT1 to:
@ 02,30 GET MINT1 PICTURE
  '999999'
```

A similar problem occurs because, QC fails to generate an 'X' format for Logical variables in SAY statements. You may have to hand-edit QC's code from:

```
@ 12,10 SAY MLOG1 to:
@ 12,10 SAY MLOG1 USING 'X'
```

in order to get this statement to work properly.

A second problem with QC integers occurs if you try to index one of your .DBF files solely, or first, on an integer field. (This means that while in fields mode, you entered a 0 in the File column for an integer variable.) The QC program .GET, in its 'Search" section, will include statements like these:

```
STORE STR(INT1, 6,0) TO MQ:KEY
  FIND &MQ:KEY
```

Because the FIND command works only with a character string, the integer-to-string function STR is used to convert INT1, before a search of the index file is executed. Unfortunately, the code written by QC doesn't work in this case. An index file with the first or only key an integer field is not correctly searched by the statements shown above. The message NOT FOUND will appear at the bottom on the run-time screen. To use an integer key, you must hand-edit quotation marks into the FIND statement: FIND '&MQ:KEY'

A final note on Quickcode: Because it is possible to commit a fatal faux pas when going from Quickscreen mode to fields mode, I routinely execute a save-to-disk operation before I enter fields mode. I don't screw up nearly as often these days as I did when first learning Quickcode, but it only takes a few seconds to write my current screen to disk. At least I can load that same screen again, after a resort to Reset if something unexpectedly goes wrong. This gives me a chance to look for the brand new stupid mistake I've made, without necessarily losing any of the editing done up to that point.

Brief notes on the dBASE editor. dBASE version 2.23B has a built-in "full-screen" editor, invoked by MODIFY COMMAND filename.typ. The chief advantage of this editor would appear to be that you can go in quickly to change a dBASE program, without having to exit dBASE. There are only a few commands, and since they are more or less similar to WordStar-WordMaster, they will seem 'natural' to many CP/M users.

I rarely use this editor myself because it lacks features I must have for efficient editing. For one thing, this turns out to be one of the dreaded Full-Screen Line-Editors, rather than a true full-screen editor. There are no provisions for fast cursor movements like MicroPro's word-forward and word-back. Global searchs and substitutions are not possible. For me, this editor's fatal flaw is the inability to support lines longer than 80 columns. Programs generated under QC which use long character fields cannot be edited in dDASE.

However, the dBASE editor does have one outstanding ability which merits comment. It won't let you load a file if there's no room for the .BAK file on the disk. If you've ever lost editing under WM because you had filled up disk A, and had no disk already logged on to drive B, you can appreciate the usefulness of this feature.

Actually, with dBASE you may be more likely to fill up the directory than the disk per se. Since dBASE doesn't have subroutines, you often find that an application requires more separate programs than you might otherwise use. This is true even if you use dUTIL or other means to merge small programs. (I use WM to 'yank' into .ADD the otherwise external files, such as .FAU, .IO, etc. This results in less disk access time when the program is run, as well as reducing 'directory clutter'.) Even if there's room on disk for the 2K (double-density format minimum sized) .BAK file, dBASE will tell you DISK FULL if the directory is full. In either case, since you can't load another file, you can never lose any editing time. ■

YOU SPENT $4,000 ON A PERSONAL COMPUTER. FOR ANOTHER $12.50, YOU CAN GET YOUR MONEY'S WORTH.

Today's personal computers have an extraordinary range of capabilities.

For a variety of reasons, however, many business people

are unaware of just how much their computers are capable of.

As a result, they aren't realizing the full potential of their investment.

THE KEY TO GREATER PRODUCTIVITY IN A WORD: SOFTWARE.

Computers do the work. Software does the thinking.

Expanding the amount of work a personal computer can do is merely a matter, then, of gaining access to a broader array of software.

And the software programs available to business and professional people number in the *thousands.*

But where do you go to find them?

THE KEY TO SOFTWARE IN A WORD: *LIST.*

LIST is the first publication that puts software first.

It contains articles by some of the most respected names in the computer field. Written to help you get the most out of your personal computer. No matter what brand it is. No matter what you need it to do.

More importantly, *LIST* contains the *LIST Software Locator,*™ a comprehensive guide to over 3,000 personal computer programs—conveniently indexed by application, industry, operating system and hardware. You'll find detailed descriptions of applications software that pertains specifically to the type of business you're in. And the type of needs you have.

LIST is sold at leading computer stores and bookstores. Or, you can phone our toll-free number (1-800-821-7700, Ext. 1110) or send in the coupon below, and receive a copy by mail. The price, exclusive of postage and handling, is $12.50.

Which, when you think about it, is a pretty small price to pay for something that can maximize a much larger investment.

LIST *is published by Redgate Publishing Company, an affiliate of E.F. Hutton.*

I'D LIKE TO GET THE MOST OUT OF MY PERSONAL COMPUTER.

Please send me _____ copies of *LIST* at $12.50 a copy plus $2.00 each for postage and handling. (Tax will be added where applicable.)

☐ VISA ☐ MasterCard (Interbank No. _____)

Card No. _____ Exp. Date ____

Signature _____

Print Name _____

Address _____

City _____ State _____ Zip _____

Send to *LIST,* Redgate Publishing Co., 3407 Ocean Drive, Vero Beach, FL 32960.

Or phone, toll-free: **1 800 821-7700 Ext. 1110**

MasterCard VISA

LIST™

The Software Resource Book For Personal Computer Users

© 1983 Redgate Publishing Company. All rights reserved.

Feature

PL/I-You're Going To Love It

by Bruce H. Hunter

PL/I is a language that you seldom hear about nowadays, particularly in the world of micros. It is one of the most powerful, expressive computer languages ever written, but it is hardly the most popular. In fact, for a language this versatile, and with the incredible programming potential it offers, it is amazing how many people have avoided learning it! There are a lot of reasons for this.

One of the reasons is that there are so few implementations for micros. I am aware of only one for 8 bit machines running under CP/M, and that is Digital Research's PL/I-80. Priced at $500, it is nevertheless a tremendous software bargain. On the other hand, at that price it is hardly an "impulse item" one casually picks up to learn "for fun."

Another reason is that PL/I was originally written for behemoth mainframes, and mainframes are available to only a small percentage of people interested in computers and computer languages today. The full set of PL/I would never fit on a micro or even a mini, because storage was never a consideration in the creation of this language; but there are the inevitable subsets, and DRI's implementation, for example, is a subset of PL/I Subset G. It is a difficult task to fit something so large into a limited space, however, and that may be why there are only a few implementations for minis and micros. With so few implementations available, many people are simply unaware that PL/I exists.

Another reason a lot of people don't eagerly pursue the learning of PL/I is that it has a formidable reputation. It is a huge language, so there is a lot to learn. It is not a friendly language like BASIC or even Pascal; when you deal with PL/I you deal with it on its formal terms! Also, it is not a very easy language to learn, partly because of its tremendous size. There are many, many rules of structure and syntax, and you must know them if you even hope to bring it up and print 'hello'. There are no books available on PL/I Subset G, so the only learning material you will find are some books on the full set of PL/I, which involves a tremendous amount of frustration since you are forced to wade through pages of technical text only to find that what you just read is not implemented in PL/I Subset G, or in any subset of Subset G that your compiler covers.

Yet another reason many people don't learn PL/I easily is that the majority of people newly exposed to PL/I, perhaps just out of BASIC or FORTRAN, have not yet had to deal with a structured language. Today we use Pascal as the "typical" structured language example, but PL/I is an ARCHETYPAL structured language since it is capable of true top-down development. Pascal has to define all functions and procedures before the program "main" encounters them, but PL/I can use true top-down structured programming by putting the main block of the program first, while secondary procedures can be placed in whatever order is clearest to the understanding of the program. There are many complex concepts involved in dealing with a structured language, but few books introduce these concepts to the novice. However, for those who persevere, the rewards are great. The combination of self-documenting descriptors, tight structure, and the most powerful set of commands, operators and functions this side of anywhere makes PL/I the exceptional, self-documenting language that it is.

For all of these reasons, and because of my love for the language, I have written a book entitled "PL/I From the Top Down." I have agreed to have *Lifelines* publish this book (a chapter at a time) over the next year, starting with this article. The scope of this book is PL/I Subset G. The subject will be dealt with informally, but from a practical point of view: the main aim is to get people up and running in PL/I. The specific purpose of this book is to provide an introduction to this language tailored for those who have never brought up PL/I or any other "fully grown" language. For further, more detailed study, I will refer you to some excellent texts already available on the full set. However, this book should get you off to a running start, and hopefully it will provide those who are interested with enough of a basic knowledge of the language to knowledgeably evaluate it and compare it with other languages. The ANSI definition of the language will be covered, and the many advantages of this much maligned language will be explored in detail. Compilers that are available for micros and minis will be discussed and compared with each other, with the full set of Subset G, and with the full set of PL/I itself. Throughout the book, parallel programming examples from several other languages will be used as a guide to the understanding of the PL/I language.

There is another reason I feel a need for a book like this. As you know, Ada is right around the corner, and with the Department of Defense sponsoring this language, it will behoove many people to become acquainted with Ada if only for the money it will pay them to do so. Soon, knowing Ada will mean never having to know unemployment. My friend Dr. William Hogan calls Ada "Pascal with a thyroid problem." Learning Pascal is a good way to be prepared for learning Ada, but Ada is going to be a gargantuan language, requiring a great deal of sophisticated knowledge about programming. Therefore, learning PL/I would also be an additional way to prepare for Ada, as PL/I is one of the the most sophisticated languages written to date. For those knowing Pascal and PL/I, learning Ada will be a relative "piece of cake." Also, Pascal, Ada and PL/I have the same "roots": Algol. It's food for thought.

The first chapter is intended to get the would-be PL/I programmer running rudimentary programs as quickly as possible. The only way to really learn a language is to dive right in and experience what it is like to program in it. My book takes a spiral approach in teaching this language, so the first chapter will simply take a superficial glance at the way PL/I handles the basic elements used in program-

(continued on next page)

ming, such as input/output, loops, declarations, etc. They will be dealt with in more detail as the book progresses. Some of you may have access to a PL/I compiler, and it will facilitate your learning if you use it in conjunction with this article, but you don't have to have one to understand what follows.

The PL/I compiler used for the writing of the programs in this chapter is DRI's PL/I-80.

PL/I

FROM THE TOP DOWN

by Bruce H. Hunter

(c) 1983

All rights reserved

CHAPTER ONE — GETTING IT UP

Your First PL/I Program

Few people have the patience to just sit down for a few months and read stacks of documentation and books on a language before writing an actual program. We all want to open the book, read a few pages and start an elementary program as soon as possible to get a feel for the language.

PL/I is a structured language. This is the first thing to bear in mind when approaching it, and there are several rules of structure to learn. Any PL/I program is a procedure (which in turn can consist of more procedures and blocks, all nested within the main procedure). So

```
Program:
        procedure options (main);
        end program;
```

is a program. It doesn't do anything, but it is a program. The program label **"Program:"** is the title or label, which all PL/I programs must have. All labels are followed by a colon. For now, everything else will be followed by a semicolon, including the last line of code. PL/I is very picky about punctuation, so note all punctuation marks carefully. The **"procedure options (main);"** tells the compiler that this is the main procedure. (It is not a macro, but it is a stand alone program. It may also call other programs and be linked with macros.) The **"end program;"** terminates the procedure, which in this case is the entire program. It is only necessary to type "end"; as that will end the procedure, but this is a very bad programming practice. Whenever you end a procedure, type "end" and then type in the label for that procedure. It would be almost inexcusable not to, because it contributes to program clarity, a must as all of you who have to maintain someone else's code know so well.

Now lets make it do something:

```
Hi:
        proc options (main);

        put list ('HI');
        end hi;
```

Notice the abbreviations. You can use **proc** for procedure. Notice the white space. PL/I is a free form language, so feel free to use as much white space as you want for program clarity and readability. The **"put list ('HI');"** will print to the screen the brief salutation. Notice the single quotes around **HI**. Before getting in much deeper, let's see how to get this little program to run. The program is written with a text editor. To play it safe, put a few carriage returns at the end of the program because your compiler must be able to read one line beyond the last line of code. It should be saved with the file extension **.PLI** (such as HI.PLI, or B:HI.PLI, if your are saving it on the B drive). The PL/I compiler will be expecting to see the PLI file extension, and it will not compile anything that doesn't have it. (NOTE: If your are saving the program files to your B drive, type **B:** directly in front of the file name, but after any command.)

To invoke the compiler type **PLI HI**, or **PLI B:HI** if you saved it on the B drive. You do not type the .PLI file extension when the compiler is invoked.

```
A> PLI HI
```

The compiler will come up, announce itself with the revision number and proceed to print "NO ERRORS IN PASS 1, NO ERRORS IN PASS 2" followed by some cryptic statistics. The compiler will have created an intermediate code program called **HI.REL**. The relocatable code is in essence an assembly program which will be linked by LINK. (PL/I's ability to create macros lies in its ability to make this RELocatable code, and the linker's ability to link it to and with almost anything, providing some housekeeping has been done.) Now type

```
A>LINK HI
```

This should start a flurry of disk activity. The linker will load, and being ready to link the grandfather of all programs, if need be, it will use disk storage instead of high speed storage to create the end program. The disk read/write heads will pop in and out, and in a minute or so, with a belch of cryptic hexadecimal stats, the linker will announce it has linked the program. Now there is a program HI.COM out on disk. The COM tells you it is a command program, which means all it needs to "run" is to have its file name entered. No extension is needed. It does not need a "run" command, nor does it need any libraries loaded. All the library functions have miraculously been linked with no effort on your part.

Before you run this program, do a DIRectory of the drive you saved it on. You should see HI.PLI, HI.REL, HI.COM, and you may see HI.SYM which is the symbol file. Now do a STAT HI.*, and it will tell it all by showing something like this:

Rec	Bytes	Ext	Asc
48	6K	1	R/W B:HI.COM
1	2K	1	R/W B:HI.PLI
2	2K	1	R/W B:HI.REL
1	2K	1	R/W B:HI.SYM

Now you are ready to run by just typing:

```
A>HI
```

That is all it takes, and the little program will prompt:

```
HI
```

You're handling strings already! Unfortunately, all the garbage you had previously on the screen is still there. It would have been nice to have cleared it off the screen. Interpreter BASIC would have done it with CLS. On the little micros like the TRS-80, BASIC is ROMmed into your machine and has control of the video map. In fact, each terminal or computer has its own code or escape sequence to clear its screen. You can usually get around this programmatically, but not always. Most books don't even bother to mention this little difficulty, or they sort of skim over it quickly to minimize it. I'm not going to do that. There are lots of "inconveniences" like this when dealing with computers because there are so few universal standards for the industry yet. So let's do it the hard way, and you'll learn more about some of the differences you'll encounter between computers in the process. My ADDS terminal wants to see a page command (form feed) or control L. (PL/I has the ability to imbed control characters into the output stream, not entirely dissimilar to BASIC's CHR$(). The caret "↑" in PL/I masks the high order nibble (four bits) of the character following it making it a control character.) A few control characters for my terminal are:

↑J	line feed	0ah
↑I	tab	09h
↑L	form feed (page)	0ch
↑M	carriage return	0dh
↑G	bell	07h

Remembering that almost all terminals are different, you will have to look up the control sequences for yours. My ADDS uses a control L (↑L) to clear the screen:

```
put list ('↑L');
```

The equivalent statement in BASIC would be

```
print chr$ (12)
```

or in Pascal

```
write (chr(12));
```

The biggest difference in the command lines is that BASIC's "print" prints to the screen only. PL/I's "put" will work with any form of stream (sequential) file output (to a disk file, to the printer, to the console, etc.). PL/I and Pascal are file oriented, and BASIC is not.

* * *

Let's discuss another concept of PL/I: precompiler commands. (A precompiler command will cause the compiler on its parsing pass to do a substitution. If a code fragment called MINIPROG.PLI exists, it will be added to the program automatically by the command:

```
% INCLUDE MINIPROG.PLI.
```

As programs get larger, this becomes an invaluable time saver, not to mention saving typos. BASIC also has an identical function called % INCLUDE which works the same way.)

The precompiler command % **REPLACE** substitutes one group of number or characters for another.

```
% REPLACE TRUE BY 1;
```

will do exactly what it says. Every occurence of TRUE will have the number 1 as its value. The immediate value of this may not be apparent, but for the sake of probability, say you have chosen to use

```
% REPLACE CLEAR by '↑L';
```

and when you move the program to another machine, you find out, as it is almost inevitable you will, that ↑L won't do what you wanted it to do. In order to get the program to run on the new machine, you have to change every ↑L with whatever will work. You only have to change it in one place in the program if you have used the % REPLACE statement! So now the program reads:

```
hi:
    proc options (main);

    % replace
        CLEAR by '↑L';

    put list (CLEAR);
    put list ('HI');
    end hi;
```

Save this, compile and link it again as you did before, and run it. Now the screen has been cleared, and the brief but friendly message has been printed. The control character did its job.

* * *

The "put" statement does not work exactly like BASIC's "print". "Print" statements assume a carriage return/line feed at the end of the line.

```
print "hot"
print "dog"
```

will result in

```
hot
dog
```

On the other hand,

```
put list ('hot ');
put list (' dog');
```

will result in

```
hot dog
```

BASIC automatically gives a carriage return/line feed pair with each print statement. PL/I takes nothing for granted and wants to be told to line feed.

* * *

Now for a look at **skip** and the **tab**.

```
put skip list ('hot dog');
```

Skip will have the system do a line feed **before** printing. It "skips" one line. To get a line feed **after** printing would take

```
put list ('hot dog↑M');
```

The embedded "control M" (carriage return) will cause the line to skip after the characters in the line have been printed. Do remember that it is the convention in PL/I to line feed BEFORE printing, however. Rely on **skip**, not ↑M.

If you want to skip several lines before printing your message, multiple **skips** are easy enough with a repetition factor after the skip. To put the string in the middle of the screen, use ↑I (the **tab**).

```
put skip (12) list ('↑I↑I↑I HI');
```

(continued on next page)

This will cause the string to be put on the 12th line and tab 3 times to somewhere around the middle of the screen.

Here it all is in this first little program.

```
hi:
     proc options (main);
          %replace
                    CLEAR by '↑L';
put list (CLEAR);
put skip (12) list ('↑I↑I↑I HI');
end hi;
```

Variables and Declarations

For those of you who are used to BASIC Interpreters, a compiler will be a different experience. PL/I is a compiler and a good one. It can't second guess what you want for variables and is rather meticulous about its housekeeping. It has few defaults and they tend to work "down," not up, going in favor of the most efficient, not convenient, so all variables have to be DECLARED.

PL/I has its own dialect as well. Strings are CHARacter. Integers are FIXED binary. Don't let binary fool you; it refers to the internal representation of the number, but in integer. Real numbers are FLOAT binary and may be decimal and/or exponential. DECIMAL is Binary Coded Decimal, the bookeeper's friend. BIT is bit. FILE is used to declare file names. There is more, but it will keep. Declarations look something like this:

```
dcl
     name char (32) var,
     dollars decimal,
     (number, nbr) fixed,
     pi float;
```

or

```
declare
     name character (32) varying,
     dollars fixed decimal,
     (number, nbr) fixed binary,
     pi float binary;
```

for the more verbose. Note that PL/I loves abbreviations. It can be "C"-like and cryptic, if you wish, but except for using available abbreviations, it's best to strive to keep it as clear as possible. In fact, PL/I can and should be elegantly clear and self-documenting. Variable names can be up to 31 characters in length, and there is no excuse for not having them be self-descriptive.

Note the varying attribute with the **char** declaration (**name char (32) varying,**). It is saying **name** is a character string of up to 32 characters, but it will accept a lot less. Care must be exercised in the declaration and use of variables. If a numeric is expected and an alpha is input instead, the program will crash. (It can be avoided.) If integer is expected and real is given, it will be truncated.

BASIC has its own declarations of sorts. It does not use the "declare" statement, but they are declarations just the same. For example, the following form is used in CBASIC-80:

```
string  name$
real    dollars
integer number, nbr
```

You will find something similar in Pascal:

```
var
     name   : string;
     dollars : real ;
     nbr    : integer;
```

The ability to declare variables lets the compiler allocate memory **before** the program is run. Storage allocated to a block is released or freed when the block is exited (unless it is declared static); therefore, it is much more memory efficient. Variables remain local to the declaring block and the same variable name can be used in any block where it is local. To those of you that have worked on a BASIC installation with only two character variable names, or in FORTRAN with six, and all of them global, you can certainly appreciate this feature.

INPUT

A lot of programming involves input. Examine the following program which contains a simple input statement.

```
instring:
     proc options (main);
          % replace
               CLEAR by '↑L';
          dcl
               name char (128) var;
     put list (CLEAR);
     put skip (6) list ('enter your name :');
     get list (name);
     put skip (2) list ('your name is ', name);
end instring;
```

The **get list** is telling the system that it is waiting for an input. It works exactly lke BASIC's "input" statement (**input name$**).

Why not simply **get** instead of **get list**? PL/I needs to know if the input is "edited" or "listed" (free form). PL/I differentiates between the two, and this will be explored further in detail.
For now, just be aware that input falls into these two categories.

```
get edit (name) (a(16));
```

This will get a name 16 characters long. If it isn't 16 characters long, it will be padded with blanks until it is.

LOOPS

Loops are another necessary element in programming. An elementary example of a loop in PL/I is the **do while** in the following example.

```
instring:
     proc options (main);
          % replace
               TRUE by '1'b,
               CLEAR by '↑L';
          dcl
               name char (128) var;
          put list (CLEAR);
          do while (TRUE);
               put skip (2) list ('enter your
```

```
            name :');
            get list (name);
            put list ('your name is',name);
            end; /*do while */
        end instring;
```

When this is compiled and linked, something happens with which someone new to PL/I should familiarize himself. You enter a first name and it takes it, but enter a full name and it reads it up to the space and refuses to read any further. What happened? The bug is called a delimiter. When the **get list** command looks for a string, it also looks for a space, comma, or carriage return to end or delimit the string. Dwell on this point for a while. Getting a single character, number, or string without a space or comma will solve this minor dilemma, and in PL/I the underscore character ("_") is commonly used to make the code readable. Here's an example of this.

Karen_Lynn_Hunter

is acceptable as a single string, or for that matter, a label.

Strings with delimiters like

William P. Hogan

will need someting else to get them entered. That **something** is EDITED INPUT.

At this point, I'm going to halt right in the middle of Chapter 1 and save the rest for the next article in this continuing series. Stay tuned for edited input, the call, blocks, and other PL/I niceties.

continued from page 27

—S0987
00 06
20 41
20 3A
20 41
20 55
20 54
20 4F
20 00
20 .
—G0000
A>A:SYSGEN
Source Disk?)press **RETURN** to skip):<cr>
Destination Disk? (press **RETURN** to skip): A<cr>
A>ERA A:TEMPSYS.COM

If you do not see the Digital Research Copyright at memory location 0990H, try lookng at 0810H, 0890H, 0910h or 0A10H. The corresponding auto-start command locations would be 0807H, 0887H, 0907H or 0A07H.

STOK SOFTWARE, INC.
Humanizing the Computer

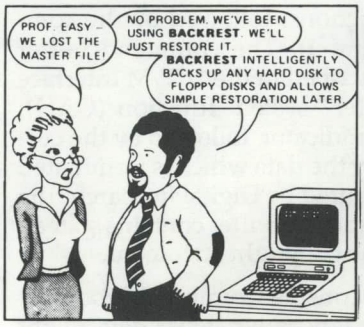
STOK PILOT™

Put your knowledge of your office environment into your computer so that your personnel will be properly guided in your absence.

STOK PILOT is a control language that allows easy development of a menu driven environment as well as an on line instructional utility for any CP/M or MP/M application. It can guide the user through an entire process without requiring the user to enter cumbersome system commands, hence making the system transparent to the user.

STOK PILOT can chain to any "COM" file program, or series of "COM" files, and regain control when the last program ends. This, and other unique features make it easy to design complete turnkey systems.

Disk and manual - $129.95. Manual alone - $14.95.



BackRest™

Hard Disk Backup, Restore and more!

- Incremental and Full backup.
- True copying of random files.
- Split large files if necessary.
- Migrate or delete selected files. **$99.95**
- Automatically restore bad files.
- Print Management reports.
- Requires CP/M 2.2, CP/M 3 or MP/M.

VISA MasterCard

THE RANDOM HOUSE ELECTRONIC THESAURUS®

$140.00

SuperDO & SuperSUB - $29.00

SuperDO allows the CP/M operator to type a string of commands that will execute one at a time. So you can walk away for a while and let your computer do its thing. Example:
A>DO ASM PROG1; LOAD PROG1; ASM PROG2; LOAD PROG2;DIR

SuperSUB is an enhanced SUBMIT command that will run on any standard CP/M 2.2 system. It runs faster than SUBMIT because it buffers the commands in memory.

Random House and the House design are TM of Random House, Inc. CP/M - MP/M are TM of Digital Research, Inc.

Dealer inquiries invited.

Stok Software Inc.
17 West 17th St.
New York, NY
10011
212 / 243-1444

Auto-Start Your CP/M

by Steven Fisher

After much head-scratching, we have come up with a way to patch the disk-resident copy of the standard Digital Research Console Command Processor (CCP) so your CP/M-80 1.× or 2.× computer will automatically run a program when first turned on. This article describes the procedure for creating a start-up command by using the utility programs supplied with CP/M-80.

A few manufacturers have provided a mechanism for invoking a user-specified program when the system is first turned on. Magnolia CP/M for Health/Zenith uses SETAUTO to allow you to specify whatever command you want; that flexibility and convenience are all you need. Some computers have been set up to always try to run a manufacturer-specified program. The Osborne-1 attempts to run a program called AUTOST; that approach leads to having the same program under two names, or having different programs with the same name. Some computer vendors allow inserting an "auto-start" command within the hardware-specific Basic Input Output System (BIOS) portion of the operating system. Having to regenerate the operating system just to change your auto-start command is an unnecessary chore, and the potential for harm rightfully scares most computer users. Many CP/M computer systems do not provide a method of automatically starting a program. But take heart, for we have a simple solution.

The Console Command Processor for CP/M-80 consists of two three-byte jump instructions, followed by the command buffer (refer to Figure 1). The command buffer follows the conventions described in the "CP/M Interface Guide" for the "Read Buffer" system function (OAH); there is a maximum length indicator, followed by the current data length, followed by the data which is terminated with a null (00H). As distributed by Digital Research, the current data length is zero, and the buffer contains sixteen spaces (20H) followed by the copyright statement.

Dr. Gary Kildall wrote the Console Command Processor to allow it to either use or ignore whatever data might already be in its command buffer. When the CCP is entered at its base, it scans the buffer, looking for a null to determine the size of its command. An empty command buffer generates the familiar "A>" prompt. When the CCP is entered three bytes past its base, the command buffer is assumed to be empty and the input prompt is displayed.

The CP/M operating system owes its widespread use to the fact that it was designed to be easily adapted by a computer manufacturer. All of the information necessary to incorporate a specific component into CP/M is contained within the Basic Input Output System (BIOS). Being hardware rather than software oriented, many disk controller manufacturers have incorrectly implemented the BIOS portion of the operating system to enter the CCP at its base for both "cold boots" (when the system is first turned on) and "warm boots" (when a program or intrinsic ends). The BIOS must have cold boots go to the CCP base and warm boots go to three bytes past the CCP base for auto-start to work properly (refer to Figure 2).

Some disk controller manufacturers support more than one disk density or format simultaneously. They often record the first diskette track in a constant density/format so their software can read the data that described the other data tracks. This mixed-mode recording is hardware-specific and not discernible by any standard software techniques, otherwise a single program could painlessly invoke auto-start on every system. All is not lost, because even mixed-mode computers have a way to read and write the system area of their disks.

Digital Research describes a GETSYS routine for loading the operating system area of a disk into memory (tracks 0 and 1 on a standard 8-inch 3740-format diskette), plus a complimentary PUTSYS procedure for writing it back. The SYSGEN.COM program contains these access methods, already customized for your disk controller's idiosyncracies. If you can format a new disk and transfer your operating system onto it, you can install autostart.

Once you have decided what your initial command line is going to be, write it down. Use an ASCII-Hexadecimal conversion chart (refer to Figure 3) to translate the characters into two-digit byte values. Write the two-digit hexadecimal length of the ASCII command line in front of the byte value of the first character (ten through fifteen are 0A-0F). Write two zeros after the last byte value to terminate the command with a null. These are the values you will insert, or "patch," into the CCP by following the procedure outlined in Figure 4. Once the patch has been completed, remove your disks and turn off the computer. After waiting a few seconds for its capacitors to drain residual current (and thereby "forget" what was in memory), turn it back on and insert the disk you just modified. The system prompt should be immediately followed by the command you installed.

If a different command appears there, your BIOS is already overlaying the memory-resident CCP's command buffer and you must disable the BIOS auto-start insertion. An auto-start command reappearing without your resetting the computer indicates that the warm boot is entering the CCP at its base, so you'll have to modify the BIOS. Not having any command appear would indicate that your disk was write protected, your cold boot does not enter the CCP at its base, or an empty auto-start command is being placed in the memory-resident CCP by the BIOS. Either unprotect your disk and start over, or change the cold boot BIOS logic.

Once your auto-start is working properly, you can change the command by repeating the patch procedure in Figure 4. Whenever you want to disable auto-start, install an empty command line consisting of a zero length and a terminating null (00 00). It's simple, when you know how.

Figure 1 — The Structure of the Console Command Processor

The SYSGEN utility program from Digital Research is used to read and write the portion of a CP/M disk that contains the operating system. The memory image of the system area of the disk is located below the memory where it would normally reside; this lower-memory version is called the "sysgen image." In a standard sysgen image of CP/M-80, the Console Command Processor occupies the memory from 0980H through 117FH. The instruction at location 0980H transfers control to a portion of the CCP that scans the command buffer at 0988H to determine the data length, which is then stored at 0987H. The instruction at 0983H transfers control to a routine that sets the data length to zero.

0980	0983	0986	0987	0988
JMP USECMD C3 5C ××	JMP GETCMD C3 58 ××	MAX 7F	LEN 00	DATA . . . 0 20. 00

Figure 2 - Proper BIOS Implementation For Auto Start

```
MSIZE   EQU   62                ; Size of system in K
BIAS    EQU   (MSIZE-20)*1024   ; Offset to minimum 2.2
CCP     EQU   BIAS+3400H        ; Base of CCP
BDOS    EQU   CCP+0B06H         ; Base of BDOS
BIOS    EQU   CCP+1600H         ; Base of BIOS

        ORG   BIOS              ; Bios begins here
        JMP   COLDBT            ; Cold Boot Entry
WBENT:  JMP   WARMBT            ; Warm Boot Entry
        JMP   CONST             ; Console Status
        JMP   CONIN             ; Console Input
        JMP   CONOUT            ; Console Output
        JMP   LIST              ; Printer Output
        JMP   PUNCH             ; Auxiliary Output
        JMP   READER            ; Auxiliary Input
        JMP   HOME              ; Track 0, Sector 0
        JMP   SELDSK            ; Select Disk
        JMP   SELTRK            ; Select Track
        JMP   SELSEC            ; Select Sector
        JMP   SETDMA            ; Define DMA Buffer
        JMP   READ              ; Read A Sector
        JMP   WRITE             ; Write a Sector
        JMP   LISTST            ; Printer Status (2.x)
        JMP   SECTRAN           ; Sector Translate (2.x)

COLDBT: SUB   A                 ; Get a 0
        STA   0004H             ; Use 0 of A:

    (hardware initialization would go here)

        LXI   H,CCP             ; point to CCP
        SPHL                    ; Stack below CCP
        PUSH  H                 ; Will go there
        JMP   GOCPM             ; Initialize page 0

WARMBT: LXI   SP,CCP            ; Stack below CCP
        LXI   H,CCP+3           ; CCP Re-entry
        PUSH  H                 ; Will go there

    (reload CCP and BDOS, and reselect curr drive)

GOCPM   MVI   A,0C3H            ; JMP instruction
        LXI   H,WBENT           ; BIOS Warm Boot Entry
        STA   0001H
        SHLD  0001H             ; Warm Boot linked
        LXI   H,BDOS            ; BDOS Entry
        STA   0005H
        SHLD  0006H             ; BDOS linked
        LXI   B,0006H           ; Default Buffer
        CALL  SETDMA

    (remove any code movng data to CCP+n)

        LXI   SP,CCP-2          ; Reset stack
LDA     0004H                   ; Get current drive
MOV     C,A
        RET                     ; Let CCP take over
```

Figure 3 — ASCII To Hexadecimal (Base 16) Conversion Chart

The Digital Research Dynamic Debugging Tool (DDT) memory-substitute command (S) requires that the new memory values be given in hexadecimal numbers. The following chart lists all of the ASCII characters likely to be used within an auto-start command:

ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX
	20	0	30	@	40	P	50
!	21	1	31	A	41	Q	51
"	22	2	32	B	42	R	52
#	23	3	33	C	43	S	53
$	24	4	34	D	44	T	54
%	25	5	35	E	45	U	55
&	26	6	36	F	46	V	56
'	27	7	37	G	47	W	57
(	28	8	38	H	48	X	58
)	29	9	39	I	49	Y	59
*	2A	:	3A	J	4A	Z	5A
+	2B	;	3B	K	4B	[	5B
,	2C	<	3C	L	4C	/	5C
—	2D	=	3D	M	4D	]	5D
.	2E	>	3E	N	4E	↑	5E
/	2F	?	3F	O	4F	‘	5F
{	7B	:	7C	}	7D	~	7E

Figure 4 — Inserting an Auto-Start Command in the CCP

By using the standard utility programs supplied with your CP/M system, you can create your own command to be used when your computer is first turned on. For instance, to have your computer automatically use the command "**A:AUTO**" (06 41 3A 41 55 54 4F 00), just type what is underscored:

A>:SYSGEN
Source Disk? (press RETURN to skip): A
Insert Source Disk in **A**, then press **RETURN** when ready:
Destination Disk? (press **RETURN** to skip): **<cr>**
A>SAVE 50 A:TEMPSYS.COM
A>A:DDT A:TEMPSYS.COM
NEXT PC
3300 0100
—D0990
0990 20 20 20 20 20 20 20 20 43 4F 50 59 52 49 47 48 COPYRIGH
09A0 54 20 28 43 29 20 31 39 37 39 2C 20 44 49 47 49 T (C) 1979, DIGI
09B0 54 41 4C 20 52 45 53 45 41 52 43 48 20 20 00 00 TAL RESEARCH

continued on page 25

dBASE II™ made easy!

QUICKCODE™

The dBASE II Program Generator

Now dBASE II is made easy with Quickcode by Fox & Geller. QUICKCODE is a program generator, a computer program which writes computer programs.

FAST AND SIMPLE

With QUICKCODE you can generate a customer database in 5 minutes. Its that fast. All you have to do is draw your data entry form on the screen. It's that simple!

NO PROGRAMMING REQUIRED

QUICKCODE writes concise programs to set up and maintain any type of database. And the wide range of programs cover everything from printing mailing labels and form letters, to programs that let you select records based on your own requirements. There are even four new data types that are not available with dBASE II alone.

YOUR CONTROL

And since you work directly with your information at your own speed and your own style, you maintain complete control. Telling your computer what to do has never been so easy.

QUICKCODE, by Fox & Geller. Absolutely the most powerful program generator you've ever seen. Definitely the easiest to use.

Ask your dealer for more information on QUICKCODE and all the other exciting new products from Fox & Geller.

FOX&GELLER

Fox & Geller, Inc. Dept. LIF 001 604 Market Street Elmwood Park, N.J. 07407 (201) 794-8883

QUICKCODE trademark of Fox & Geller, Inc
dBASE II is a trademark of Ashton-Tate

```
HPINPU* 0114   HPOUT2* 0123     NFACT* 011D
3 Undefined Global(s)          <— L80 also tells us that three labels
                                   are external
*B:HPIMATH/S                   <— Tell L80 to search the math library
*/U                            <— Ask for a memory map and list of
                                   any undefined globals.
Data    1000  165E
Program 0100  0644             <— L80 obliges with a memory map,
                                   no undefined labels
*B:TSTFAC/N/E                  <— Tell L80 to save program as
                                   "TSTFAC.COM" and return to
                                   CP/M
Data    100   165E
Program 0100  0644
[0000   165E  22]              <— L80 shows memory map, total
                                   program length, and number of
                                   sectors saved (22).
AO>                            <— Back to CP/M
```

; Test of the FACTORIAL function. ***Listing 5***

```
; 07/23/82          by Thomas Hill
; system equates
CPM     EQU     0
BDOS    EQU     CPM+5
PRTBUF  EQU     9
CR      EQU     ODH
LF      EQU     OAH
; declare external modules
        EXTRN   HPINPUT         ; input module
        EXTRN   HPOUT2          ; output module
        EXTRN   NFACT           ; factorial module
        CSEG
TSTFAC: LXI     D,SIGNON
        MVI     C,PRTBUF
        CALL    BDOS            ; say what we are doing
LOOP:   LXI     D,ASK
        MVI     C,PRTBUG        ; ask for number
        CALL    BDOS
        LXI     D,IN$NUM        ; input value storage
        CALL    HPINPUT         ; go get it, tiger!
        CALL    CRLF            ; new line
        LXI     D,IN$NUM
        CALL    NFACT           ; form the factorial
        LXI     D,IN$NUM        ; factorial now at input
                                ; storage
        CALL    HPOUT2          ; output it
        JMP     LOOP            ; do it forever.
CRLF:   LXI     D,CRLF$MSG
        MVI     C,PRTBUF
        JMP     BDOS            ; print a CR,LF at
                                ; console and return
; declare data area
        DSEG
SIGNON: DB      'Test program for high precision integer math
        DB      factorial'
CRLF$MSG:
        DB      CR,LF,'$'
ASK:    DB      'Enter a number: $'
; value storage
IN$NUM: DS      128
        END
```

Macros;

Symbols:

```
ASK     0039"  BDOS    0005  CPM     0000  CR      000D
CRLF    0028'  CRLF$M  0036" HPINPU  0014* HPOUT2  0023*
IN$NUM  004A"  LF      000A  LOOP    0008' NFACT   001D
PRTBUF  0009   SIGNON  0000" TSTFAC  0000'

No Fatal error(s)
```

Sample run of TSTFAC program: ***Listing 6***

```
A0>B:TSTFAC<cr>
Test program for high precision integer math factorial
Enter a number: 45 =      <— try 45. Note that input is terminated by '='.

119,622,220,865,480,194,561,963,161,495,657,715,064,383,733,760,
000,000,000
                          ↑— program displays answer to 45!
Enter a number: 32 =

26,313,083,693,369,353,016,721,812,160,000,000  <— answer to 32!

Enter a number: 99 =      <— Let's try a big one

933,262,154,439,441,526,816,992,388,562,667,004,907,159,682,643,816,
214,685,929,638,952,175,999,932,299,156,089,414,639,761,565,182,862,
536,979,208,272,237,582,511,852,109,168,640,000,000,000,000,000,000,
000
                          ↑— anybody care to check this?
Enter a number: ↑C        <— return to CP/M
```


FORTH-79

Version 2 For Z-80, CP/M (1.4 & 2.x), & NorthStar DOS Users

The complete professional software system, that meets ALL provisions of the FORTH–79 Standard (adopted Oct. 1980). Compare the many advanced features of FORTH–79 with the FORTH you are now using, or plan to buy!

FEATURES	OURS	OTHERS
79-Standard system gives source portability.	YES	_____
Professionally written tutorial & user manual.	200 PG.	_____
Screen editor with user-definable controls.	YES	_____
Macro-assembler with local labels.	YES	_____
Virtual memory.	YES	_____
BDOS, BIOS & console control functions (CP/M).	YES	_____
FORTH screen files use standard resident file format.	YES	_____
Double-number Standard & String extensions.	YES	_____
Upper/lower case keyboard input.	YES	_____
APPLE II/II+ version also available.	YES	_____
Affordable!	$99.95	_____
Low cost enhancement options; Floating-point mathematics Tutorial reference manual 50 functions (AM9511 compatible format)	YES	_____
Hi-Res turtle-graphics (NoStar Adv. only)	YES	_____
FORTH-79 V.2		$99.95
ENHANCEMENT PACKAGE FOR V.2: Floating point		$ 49.95
COMBINATION PACKAGE (Base & Floating point)		$139.95

(advantage users add $49.95 for Hi-Res)
(CA. res. add 6% tax; COD & dealer inquiries welcome)

MicroMotion
12077 Wilshire Blvd. # 506
L.A., CA 90025 (213) 821-4340
Specify APPLE, CP/M or Northstar
Dealer inquiries invited.

Feature

Thunderclock Routine

by David W. Walker

As a member of the small but growing fraternity of Apple CP/M users, I was very pleased to see Matthew Von Maszewski's piece in the March 1982 issue of *Lifelines*, containing a program applicable specifically to the Apple (and, amazingly, even more specifically to the Apple with a Sup'R'Term 80-column card—a combination of equipment that I, too, happen to enjoy).

Following that salutary precedent, I offer herewith a short program, which applies to a similarly limited subset of CP/M users (specifically, users of Apple CP/M with the Thunderclock peripheral card). For Apple CP/M users without a Thunderclock, it offers an example of calling routines on a peripheral card from the CP/M environment, which can be adapted to other brands of clock card or to other kinds of peripheral cards.

Like most peripheral cards for the Apple II, the Thunderclock contains a firmware control routine to facilitate the process of reading the current date and time from the clock registers (as well as some other functions, such as setting the clock, that I have not attempted to use in the enclosed program). The entry addresses for the firmware routines depend on the Thunderclock's physical location — which of the Apple's peripheral slots it is plugged into. In slot 4, for example, the clock firmware appears at addresses $C400-$C4FF. Because Microsoft's Z80 Softcard very cleverly translates the "normal" Apple memory addresses in order to get the contiguous block of memory the CP/M expects, the "Apple" addresses $C400-$C4FF would appear to the CP/M environment as E400H-E4FFH. If the clock card were, instead, in slot 5, the firmware would appear at $C500-$C5FF (E500H-E5FFH); and so on.

Reading the Thunderclock from a 6502 assembly language program is a fairly simple matter. The calling program needs simply to place a specified "control" character in the 6502 accumulator (to tell the firmware in which of four available formats to return the current date and time), call the "write" routine at $Cn0B (where n is the clock's slot number), then call the "read" routine at $Cn08. The firmware returns a string in the specified format, beginning at address $200 (or F200H, to CP/M). That string can then be moved somewhere else for other use or, as in my program, sent to the console using the "print string" function provided by CP/M.

The area from $200-$2FF is reserved as an input buffer in most of the languages that run on the Apple II. Fortunately, Microsoft's CP/M implementation has enough free space at the corresponding locations (F200H-F217H is all we need), so that the clock routine doesn't clobber anything by putting the string there. If you have patched any of the I/O vectors, however, you might be using the area beginning at F200H for something important. In that case, you would add to my program a routine to save the contents of F200H-F217H before calling the clock write routine and to restore those contents after printing the date/time string.

Of course, the Thunderclock firmware routines are in 6502 code. Executing them directly from CP/M would produce decidedly unwanted results. Microsoft, however, has thoughtfully provided Apple CP/M users with a straightforward way to transfer control to the Apple's native 6502 CPU temporarily, to execute routines in 6502 code. The method is simply to store the entry address of the 6502 routine in the location F3D0H (reserved for that purpose), then write something (anything) to the first address occupied by the Z80 Softcard itself. Like the Thunderclock, the Z80 card will occupy a block of addresses beginning at En00H, where n is the number of the card's peripheral slot. Since, like the Thunderclock, the Z80 card might (at least in theory) be in any slot from 1 to 7, Microsoft's CP/M implementation very thoughtfully stores the card's first address at F3DEH-F3DFH, so you can find it. Therefore, the standard technique for calling a 6502 subroutine is to store the routine's address (in the form that the 6502 will recognize) at F3D0H, load the Z80 card address from F3DEH into the Z80's H,L register pair, and write to that address by executing MOV M,A (in 8080 code) or LD (HL),A (in Z80 notation).

In this case, we also want to pass a "control" byte to the 6502, since the Thunderclock's "write" routine expects such a byte in the 6502's accumulator. Again, Microsoft has anticipated the need, and has provided a way to pass data to any or all of the 6502's registers, by storing the data in a "pass" area of memory beginning at F045H. The byte contained at F045H will be passed to the 6502's accumulator before calling the subroutine pointed to by F3D0H; the byte contained at F046H will be passed to the 6502's Y register; and successive bytes will be passed to the 6502'2 X register, its processor status register, and its stack pointer, respectively. In my CLOCK program, only the accumulator data is significant, so we don't care what may be at locations F046H-F049H.

Like the Softcard, the Thunderclock card might in theory be plugged into any of slots 1 through 7 in the Apple II. It is simple to call the clock's "write" and "read" routines if you know what slot it will be in. You may decide to move the clock, however; so it's better to take the slight extra trouble to start with a routine that finds what slot the clock is in, and uses that information to set the call addresses passed to the 6502. Finding the clock is straightforward: compare the bytes contained in the first three addresses associated with each slot to the known first three bytes of the clock card, until a match is found. (If no match is found, then the Thunderclock isn't there, and the routine can avoid attempting to call a non-existent firmware routine -- which would almost always result in a fatal "hang" of the system.) The Thunderclock's first three bytes are 08H, 78H, and 28H, in order. As far as I know, no other firmware card available for the Apple][has those first three bytes.

The clock finder routine that I have used starts with slot 7 and moves downward, exiting when the clock has been found or all the slots from 7 to 1 have been checked. If the clock is missing or is in a lower-numbered slot than the Z80 Softcard, the finder routine will access the Softcard's address, the location that turns off the Softcard and transfers control to the Apple's 6502. Once again, Microsoft has anticipated our needs and avoided problems. Only a "write" operation to the Softcard's address will transfer control to the 6502. A "read" operation to that address will simply return a byte of data.

As I mentioned above, the Thunderclock firmware can return the current date and time in any of four formats. I have elected to use only one—the format THU NOV 11 8:35:24 AM—to keep things simple. The Thunderclock operator's manual explains the other formats quite clearly; and it is simple to substitute any of the others by substituting the appropriate control character for the "/" that I have used, using the appropriate string length in calculating where to put the "$" string terminator before calling CP/M's string output routine, and using the appropriate starting address for the string. (The string returned in response to the "/" control code begins with a quote character, so I begin printing the string at the second character, to skip the quote. Other formats, for reasons related to the way that Apple's two BASICS interpret the input buffer contents, do not begin with a quote character. If you were using one of those formats, you would naturally begin printing with the first character.)

The Microsoft Z80 Softcard contains a Z80 CPU, and can execute programs written in Z80 code or in the more common 8080 subset. I have written the enclosed program in 8080 assembly language, so that it can be assembled using the ASM program that comes with the Softcard. The textfile can be prepared using the supplied ED utility, which is adequately if not brilliantly explained in the documentation supplied with the Softcard. For the new user, coming from Applesoft, Pascal, or almost any of the text editors available for the non-CP/M Apple, ED will appear strange and obscure; but it can be mastered with a little practice. Take my word for it. When the textfile has been entered and saved to disk — using, say, the file name CLOCK.ASM — then it can be assembled by typing the command ASM CLOCK, and converted to an executable COM file by typing LOAD CLOCK. Then, typing CLOCK from the CP/M command level will run the program and display the current date and time on the console screen. Nothing to it, right? ■

Feature

A Review of Alpha Software's Data Base Manager

by Ron Watson

The computer industry is probably the most buzz-word prone area of human endeavor. Surely, there must be a small, secret group of public relations experts hiding in the sewers of San Jose, or camped out in a wilderness area near Seattle or White Plains, who spend all their waking hours devising new terms to be infiltrated into the lexicon of computer professionals. Unfortunately, they provide only the terms, allowing the definitions to be made up to suit the purposes of the ones who use them. The buzzwords sprout like dandelions in bloom to have their seeds spread by the winds of the media to every corner of our field. Many are used only briefly and then fade away, forgotten. But some take root and become permanent members of the language. After a while, everyone uses the successful ones casually, with the full assurance that the listener understands even if the speaker is not too certain of the meaning. Sometimes the new term itself flowers and gives birth to derivative terms which add even more respectability to the parent.

Such a term is "data base." It has been with us quite a while by computing standards, well over fifteen years. It has spawned 'data base management system," "data management," "DB/DC," and many others, as well as various new job titles in large organizations and, incidentally, a data processing specialty. I would expect such a ubiquitous term, one that we all use so often, one that George Fenniman would have said was 'heard around the house every day', I would expect such a term to have a good, solid, well understood definition. Not so. The definition may exist; it is certainly not well understood. As proof of this, the term can be found describing such completely different products as dBase II and Alpha Software's 'Data Base Manager,' the later being the subject of this review. Alpha's product may serve to lend precision to the term "data base" by acting as an example of the absolute minimum capability required to qualify for its use.

But first the good news. Data Base Manager is well packaged. The manual is in the now familiar nine by eight inch binder used by IBM and comes with plastic inserts to hold the master program and demonstration diskettes. A tutorial audio cassette is also included. Except for a slightly unpleasant type font, the documentation is well produced, and well written.

The text is written assuming the reader is completely unfamiliar with the hardware, and is very consistent in this assumption. It is organized in tutorial fashion, and so does not function very well as a reference manual. After I had become familiar with how to use the software, I had difficulty locating answers to specific questions in the manual.

The tutorial cassette is meant to be heard while one uses the program with the demonstration data diskette provided. This is well done, properly timed, and quite effective.

What we have here is something that looks like a Ferrari and drives like pushcart.

The first procedure one is asked to do is back-up the master disk. A .BAT file is provided for the purpose. It runs a program that will make two copy-protected master disks from the original. For some reason known only to Alpha, this procedure uses just one disk drive (the system

requires two to function at all), thus requiring the operator to alternate master and copy in and out of drive A thirty or forty times. Since the master can not be write-protected during this procedure, there is some danger of getting hands crossed and destroying the master.

The normal start-up procedure is to place a master disk in drive A and boot the system. This brings up a request to insert a data diskette in drive B. If the data diskette contains a data base file, then it is used. If no data base file is found, the file initialization menus are brought up, and we encounter a limitation: only one file of data per diskette.

The initialization procedure allows the user to name up to 19 fields of up to 24 characters each. A field name may contain no more than ten characters. No data type may be specified; all fields are alphanumeric by definition. Once the definition is finished, only the field names can be changed. No facility is provided to reformat or re-arrange the data once defined.

Incidentally, all operator input is apparently programmed with something very much like the BASIC "Input" statement: no commas or colons can be entered. No error recovery is provided, so entering a comma gives one the BASIC message 'redo from start.' The initialization screen suggests that the operator press the num-lock and caps-lock keys; the first because none of the cursor control keys on the keyboard are used, the second because prompt responses require upper case input. If they wanted to simplify the program by eliminating code to shift lower case to upper on command responses, they should have set these keys programmatically, a simple procedure on the IBM-PC.

The main menu screen contains a copyright notice and the author's name, the data base name assigned when the file was initialized, and a list of eleven items from which to choose.

Item #1 allows the user to enter new data. The program puts up a screen showing one field per line in the same order as they were defined. Data is entered from top to bottom, one field at a time, terminating each field with the enter key. A semicolon entered as the first character of a field causes the cursor to return to the previous field. A slash may be used to duplicate data from the previous record. When the last field is complete, the operator has options to add the record to the file, make changes, return to the menu or bypass the record. Adding a 120 character record took 6.2 seconds and seemed slow. An inadvertent comma causes the BASIC message "redo from start" to display on the screen line following the line in error, making a mess of the display. If too many characters are entered, the field is cleared and the cursor positioned to the first field position, but no beep is sounded. The input procedure was apparently designed with more concern for the ease with which it could be programmed then for the ease with which it could be used.

Returning to the main menu, selecting item #2 allows the user to view the data. This brings up a sub-menu with three selections: view all data, view a range of records, and view a single record. All three options present a screen similar to the data entry screen; the first shows all the records in the order in which they were added to the file, the second and third options allow the user to enter a record number or range of record numbers to be displayed.

The real lack of any sophisticated data structure begins to become apparent. One may only view data at this point if an arbitrary value, record number, which has nothing to do with the data from a user's viewpoint, is known. The seriousness of this deficiency becomes more apparent in the examination of subsequent data presentation options.

Item #3 on the main menu, is labelled 'SORT DATA,' and it is bound by this system to item #4, 'SEARCH DATA.' Before either procedure can be used, the user must have defined a report using menu item #8, 'FORMAT REPORTS.' This restriction applies because the only thing that can be done with sorted or searched data is to prepare a report. Sorted data can not be searched, but then searched data can not be sorted, either.

The file can be sorted on the first five characters of one field in either ascending or descending sequence, using either numeric or alphabetic logic. Since there is no provision to define a field's content as either numeric or alphabetic, the sort procedure will ask the user which is to be done. A fifty record file took a minimum of 21 seconds and a maximum of 31 seconds to sort; the reason for the variation was not apparent. When the sort is complete, the operator is shown a list of the defined reports, and asked to select one to be used to display or print the data. The report procedures are discussed below.

Item #4, search data, is really quite powerful compared to what we have seen so far, though it would more properly be called 'select data'. There is a three field search with logical operators, a "wild card" search that will scan a field for a substring, a matching field search, and one called Soundex, which will look for phonetic equivalances. Sound pretty good? Keep reading.

Subitem #1, the three field search, requests the user to select the field number to be searched from the list of defined data items, then the logical operator to be used and finally the contents to be compared. This is repeated once for each field to be searched. The search will be done for those records that meet all the selection criteria entered, though this is not made clear in the text. There is no way to use the "logical or" connective.

Subitem #2, the "wild card" search, will test a field's contents for the substring entered by the user. No mention is made in the manual about upper/lower case difficulties, and I didn't test to see what might or might not work. This option allows only one field to be searched for one string at a time.

Subitem #3, search for records, is the only way to look at a group of records without having a report format defined. The search is done for all records with a particular field matching the constant which is entered in response to the prompt.

Subitem #4, the phonetic search, will select records with a field contents that is phonetically equivalent to the input data. Though not tested extensively, this seems to function as promised.

So, we have all these ways to select records from the data file; what can we do with it once it is selected? Why, we can make a report, of course, but that's all. Well, maybe

that's enough. Maybe a really fine report generator will make up for the limitations observed so far.

A report is created by selecting item #8 from the main menu, 'FORMAT REPORTS.' The user is first asked for a name to be given to the report. A maximum of ten reports may be defined. The name "LABEL" is special in that a mailing label format is generated instead of a report format. Speaking of names, the manual makes no mention of the need to keep the report names unique, but the name entered becomes the name used on a file containing the report specifications which is written on the data diskette. Using the name of a report that already exists without first requesting deletion of the original name seems to create considerable confusion for the program. The problem can be resolved easily enough by an experienced user: he can use DOS to delete the specification file from the data diskette. This bug is not as serious as it sounds: how often would someone give two reports the same name? The really serious problem with the reporting capability is that there is so little.

The method of specifying the report is simple enough. The user does have to know the total number of fields to be printed before he starts, and as there is no list of fields shown to him while he specifies the report. He also has to know what numbers have been associated with his data fields. Once the report has been defined and tested, there is no way to change it except by respecifying it entirely. But that's not so serious, nearly everyone will get it right the first time, and for those who don't, retyping the specification is an appropriate punishment.

I had considerable difficluty understanding what could be done with the formatted reports. Part of the difficulty stemmed, no doubt, from the disdain I had accumulated for the package up to the point I began to experiment with this feature. Perhaps I did not give it a fair test. After twenty years of struggling to design reports that are informative and esthetic, I have little tolerance for the amatuerish results available with this program. The data could be listed, that was clear, but there was no way to truncate a field on the report. No editing of numeric fields was possible. There was only one control break available, and the method of requesting it was obscure. Some totaling was possible, but only one calculated field could be printed. It seemed everything I wanted to do was impossible or yielded unsatifactory results.

There is an automatic tabulation option that allows the user to have the program determine the column positions for each field and then change the generated columns afterwards. The specification procedure is simple but slow, and as mentioned above, changes can only be made when the report is originally specified. One could learn to live with these shortcomings, I suppose, by making printouts of the record layout and report format specifications to be referred to while working with the report generator. A preliminary sketch of the report would also be a good idea as the trial and error method is made too time consuming by the inability to edit report specifications after the results are seen.

Conclusion

I have tried to imagine some situation where this program might be of some use to someone. But any application simple enough to be practical would be better done with a small metal box of 3 × 5 index cards. Even the greenest, most timid new user will quickly outgrow this package.

Recent issues of PC magazine have carried a two page, two color ad that promotes this package as the solution to everyone's data management problems. Well, all the ads in world will not create usefulness where none exists. Alpha Software would be well advised to rearrange their corporate budget so as to allow more money for design, even if they have to reduce the amount spent for promotion. ■

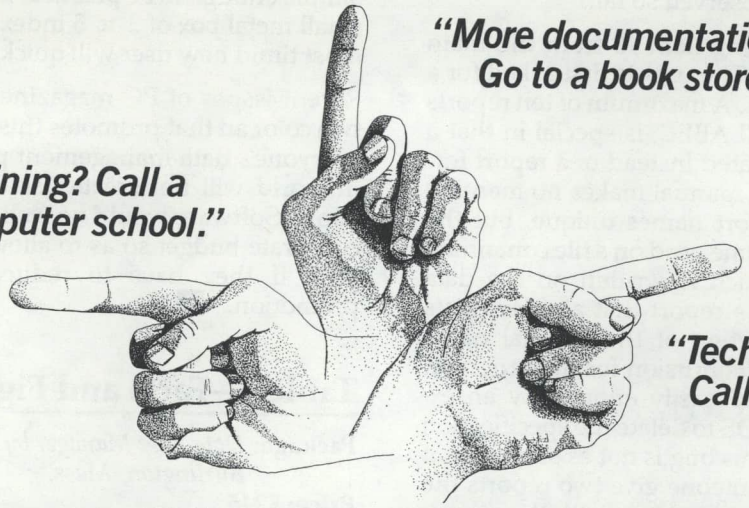
Table 1—Facts and Figures

Package: *Data Base Manager by Alpha Software Corp., Burlington, Mass.*
Price: *$245*
Available for:
IBM Personal Computer.
Required Supporting Software:
PC DOS.
Memory Requirements:
Unknown minimum. 64K max.
Diskette Capacity Required:
Two drives. Will use single or double sided. Data files restricted to one diskette. program must remain in drive A.
Utilities provided:
Program backup.
Record size and type of limitation.
Internal storage is fixed length ASCII with carriage return and line feed.
Maximum of 19 fields, maximum of 24 characters per field.
Report lines up to 132 characters.
User skill level required:
All documentation and procedures assume no user familiarity with either hardware or software. Audio cassette included that steps a new user through a tutorial file that is also included.
System upgrade policy:
Not known.

Table 2-Qualitative Factors

Documentation:	
Organization for learning	7
Organization for reference	5
Readability	6
Includes all needed information	4
Ease of use:	
Initial start up	3
Conversion of external data	NA
Application implementation	3
Operator use	4
Error recovery:	
From input error	4
Restart from interruption	4
Restart from media damage	4

(continued on next page)



Interested in dBASE II™ or 1-2-3™? Beware The Dreaded Finger Pointers!

Sound familiar? Does your dealer turn into a "finger pointer" when you need help?

At SoftwareBanc we offer a complete system that doesn't stop when your software is delivered.

Careful Product Selection

Do you get bewildered by the endless lists of software you find in most ads? Let us be your quality control department.

We only sell the best programs on the market. After a thorough evaluation we chose dBASE II™ for data processing, and 1-2-3™ for financial management.

Our complete line of add-on products help you to continue to get the most from your software.

Expert Technical Support

When you buy software from us, you can rest assured that help is only a phone call away. Just call us at (617) 641-1235 for all the free support you need.

Free dBASE II™ User's Guide

Order dBASE II™ from us, and you'll receive a free copy of our dBASE II™ User's Guide. You can also buy the User's Guide first for only $29, and then receive a full credit when you buy dBASE II.™

French Translation
La Commande Electronique
5 Villa Des Entrepreneurs
75015 Paris, France

Japanese Translation
JSE Int'l
9F Toyo Bldg. 6-12-20 Jingmae
Shibuya-ku Tokyo, Japan 150

Free 1-2-3™ Utility

1-2-3 TRANS is a menu driven program that will quickly and easily transfer files from dBASE II™ to 1-2-3™ and back again. Free with 1-2-3™ purchase!

1-2-3™ & dBASE II™ Classes

Want more in-depth information about dBASE II™ or 1-2-3™? Attend a SoftwareBanc Seminar near you. Each session runs from 9 to 5, and costs $175.

Los Angeles July 18-22	***Anchorage*** August 11-12
Washington, D.C. Aug. 29-Sept. 2	***New York City*** September 19-23

Prices You Can Afford

†1-2-3™	Call for price*
†dBASE II™	$479
†ABSTAT™	$379
dBASE II™ User's Guide	$29
DBPlus™	$95
dGRAPH™	$199
dUTIL™	$69
dNAMES™	$109
QUICKCODE™	$199
TEXTRA™	$70*

*Only available for IBM PC with MS-DOS.
†No-risk 60 day money back guarantee

Free Catalog

If you want to learn more about SoftwareBanc, call or write for our free product catalog.

SoftwareBanc

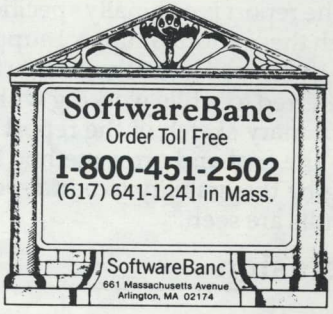
661 Massachusetts Avenue
Arlington, Mass. 02174

To order call: (800) 451-2502
(617) 641-1241 in Mass.

For technical support call:
(617) 641-1235

™Manufacturer's trademark

Payment may be made by: MasterCard, Visa, check, C.O.D., money order. Mass. residents please add 5% sales tax. Add $5.00 for shipping and handling.



Macro of the Month

Michael Olfe

This macro constructs a key redefinition file for an MS/PC DOS 2.0 which uses the ANSI.SYS device driver. The new key definitions take effect as soon as the created file is TYPEd from the DOS command level, or dumped to the screen by a program which uses function calls 2,6, or 9 for screen output. The key redefinition file is an ASCII file containing escapes, and can be edited like any other ASCII file. If the named file exists, you have the option of editing it or using another name. To help humans read the key definition files, a header on each line gives the key name and the redefinition in ASCII.

Unfortunately, the redefinitions will not be recognized by PMATE under DOS2.0, although the macro and PMATE itself run fine under 2.0. Other editors, like EDLIN and Final Word, will recognize the redefinitions.

This macro calls a permanent macro called "i," which gets a string in response to a prompt. "I" is defined as:

```
qa[g±Aa$@k=127 [-d±][@k=13[/]@ki]]

[b9kb9et.iName of key definition file?$
#b8m @f±A@8${gFile exists. Want to edit it (Y/N)? $
(@k="y)!(@k="Y){btexf±A@8$/}{__}]

b8e b9k b6k b6e
 create a look-up table for ALT codes
    iQWERTYUIOP     ASDFGHJKL     ZXCVBNM$
b9e

[0gType key to be redefined or ESC to stop$qn$
 @k=27__ iKey: $
 @k>127{@k-128v0
    ((@0>15)&(@0×51)){iALT $b6e a @0-16m @tv2 b9e
    @2i}
    ((@0>103)&(@0×114)){iALT F$@0-103½}
    ((@0>119)&(@0×132)){iALT $@0½}
    ((@0>83)&(@0×94)){iSHIFT F$@0-83½}
    ((@0>93)&(@0×104)){iCTRL F$@0-93½}
    ((@0>58)&(@0×69)){iF$@0-58½}
    }{@ki0v0@kv1}
(20-@x)[i $] iDefinition: $
b7k b7e .iType definition$
b9e b7g
(80-@x)[i $] 27i 91i @0>0{i0$ @0½}{@1½} i"$
 b7g i";13p
$qr]

axo±A@8$bte
```

(continued from page 2)

34) Matrix printers would replace formed character printers

35) S100 machines would continue forever

36) The five inch screen on the Osborne was adequate

37) Apple II would be replaced by the Apple III (which is rumored to be discontinued).

38) Apple's future is dependent upon the success of Lisa.

39) IBM had missed the PC market.

40) The IBM-PC represented a quantum leap in technology

41) Softcards would disappear from the marketplace within a few months of their release.

42) Microcomputers will take jobs away

43) Documentation is read

44) The "----------" language makes complex programming tasks trivial (you fill in the blank with your favorite programming language).

45) CP/M-80 programs would run under CP/M-86 without modification

46) CP/M-80 assembly language programs could easily be translated and run in sixteen bit environments.

47) Programs translated/re-compiled for sixteen bit machines would be faster and more efficient than their eight bit counterparts

45) And on, and on, and on . . .

I suppose that the best indicator of the validity of such allegations is that their propagators are for the most part no longer to be found anywhere.

It is in many respects sad that so many have been led astray by such irresponsible and capricious claims of a few. Sadder still is the loss of hundreds of thousands of person hours which have resulted from placing faith in such beliefs.

Anyway, I guess I had better go . . . in the immortal words of Woody Allen "I'm due back on Earth."

Software Notes

New Products

WordPlan

Idea Software
PO Box 968
Fremont, CA 94537

This text and data formatting package allows easy manipulation of textual and numeric values within documents. It solves business problems where numbers are calculated values and both numbers and text change frequently. WordPlan lets you set up your template document and embed the necessary variables and equations. Then all you have to do is change one variable to change all of the associated equations and variables. The number of numeric variables you can have is 200. You can also use up to 27 text variables. WordPlan accepts text from any text editor.
Requirements: CP/M

Price: $195

dBASIC

Active Software Marketing
1953 E. Apache
Tempe, AZ 85281

This Indexed Relocatable Library for CB/80 and CB/86 compilers provides direct access to dBASE II files. CBASIC programs can read or write any .DBF or .DBR database in random or sequential mode and can read, modify or create file structures in native dBASE format. Many dBASIC functions are direct translations of dBASE II commands. "SELECT PRIMARY" is "CALL SELECT," "USE MYFILE" becomes "CALL USE ('MYFILE')," "SKIP-8" translates to "CALL SKIP (-8)" and "COUNT TO XYZ" is the same as "XYZ/=COUNT." APPEND, DELETE, and RECALL are also included. Up to nine files may be open at the same time and new functions allow direct retrieval of database characteristics such as filed names and types.

Price: $395

FILER

COMPU-DRAW
1227 Goler House
Rochester, NY 14620

This program compacts, archives and catalogs disk files. For applications that result in a large number of relatively small files (6K or smaller), FILER can result in disk space savings of several hundred percent. FILER also archives and catalogs files with features including date-stamping and descriptive text. Filer enables the user to save up to 1000 files of any type in a single file called a Volume. The user may describe each file by over 100 characters of descriptive text. If updated versions of a file are saved in the same volume, FILER automatically assigns them version numbers. The files in a volume may be cataloged on the console or printer or into a disk file. Catalog listings may be sorted by name, extension and/or version attributes. FILER is completely menu- and prompt-driven.

Requirements: CP/M-80 (8080 or Z80), 56K. Also runs under CDOS.

Price: $49

FILEBASE

EWDP Software Inc.
POB 40283
Indianapolis, IN 46240

This program processes files of records comprised of comma delimited fields. It is menu and prompt driven. Options include record selection, merging, sorting, creating new files, appending to existing files, selecting out a subset of fields and modifying their contents, and rearranging fields or adding new ones. Records can be selected or excluded by testing field contents with a comparator or against a list of up to 300 user entered values. Other select/exclude methods include a range or list of up to 1000 record numbers or by interactive examination of each record in real-time. During the interactive process, records can be printed one field per line with page ejects between records or with any number of lines between them. Files can be combined into one new file or divided into two files. FILEBASE can convert variable length records to fixed length or certain types of fixed length records back to variable length.

Requirements: CP/M (Z80), 64K

Price: $75

New Books

Hayden Book Company Announces "Secrets of Better Basic"

Secrets of Better BASIC reveals the sophisticated programming tricks and techniques used by professional software authors for writing faster and more effective BASIC programs.

Written by Ernest E. Mau and published by the Hayden Book Company, Inc. of Rochelle Park, New Jersey, the book also offers faster and more effective programs for testing and debugging programs, more efficient use of memory, string-handling, using loops and subroutines, and creating disk files.

The book also includes five appendices that include the ASCII codes and equivalents, numerical systems and conversions, some "BASIC" functions, sample disk and memory tests, and some useful software.

Price *$14.95*

Using Microcomputers in Business: A Guide for the Perplexed, Second Edition

Using Micrcomputers in Business. . . is a background reference for any purchaser of a micrcomputer system or software for a small business.

Written by Stanley S. Veit and published by the Hayden Book Company of Rochelle Park, New Jersey, the book describes the advantages and disadvantages of computerization and enables the potential user to make intelleigent purchasing decisions.

It explains business applications from the fundamentals of microcomputer systems to the fine points of word processors, accounting programs, data bases, and disk drives. New chapters focus on the value of electronic spreadsheet programs and microcomputer networking.

Price: *$12.95*

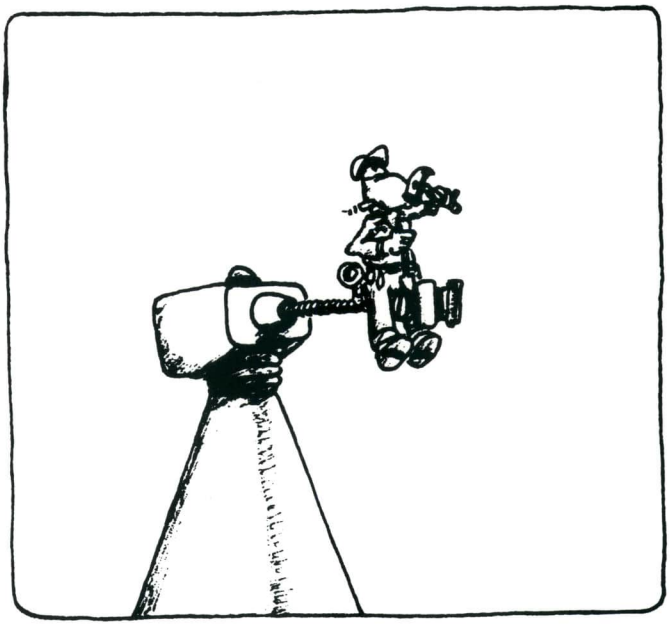
New Versions

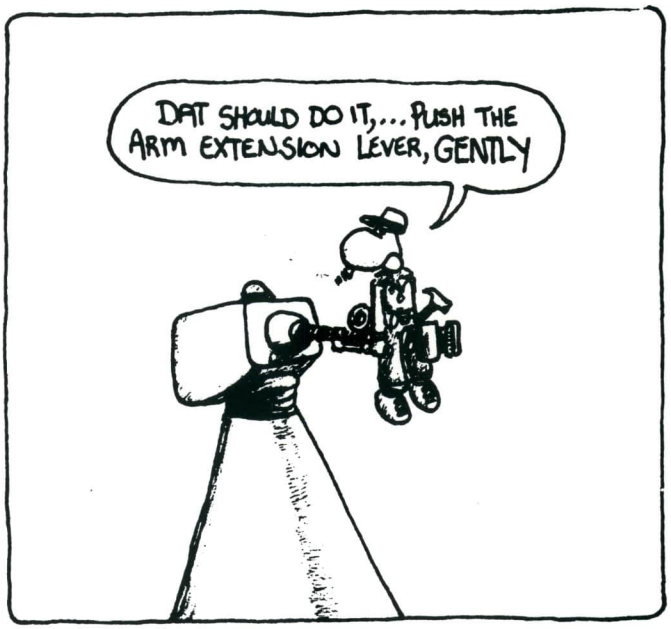
NEW VERSIONS

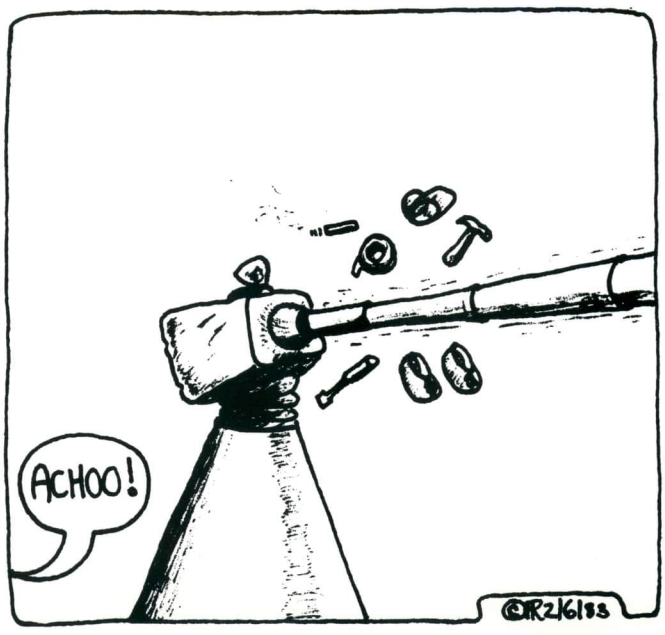
BOSS v1.19
BSTAM-86 for VICTOR 9000
(CP/M-86 & MDOS)
C-FOOD SMORGASBORD v1.3
D-BASE-II v2.40
PAS-3 Medical v1.92
PAS-3 Dental v1.79
Precision BASIC-86 v1.6
(for MS-DOS)
STIFF UPPER LISP v3.1
WORDSTAR SPANISH v3.0
Lattice "C" Compiler v1.04

Attention Subscribers

Lifelines would like to announce the reorganization of its staff. We plan to include the expirations date on the mailing labels as soon as our new system is implemented. Look for reviews on T/MAKER III, Microsoft FORTRAN, Supersoft FORTRAN, the Wedge, Prospero Pascal, more on PL/1-80, and exciting new columns on MSDOS, etc.







Lifelines™/ **The Software Magazine**
1651 Third Ave., New York, New York 10028

Second Class Postage Paid
At New York, N.Y.

EXPIRATION DATE: 12/87