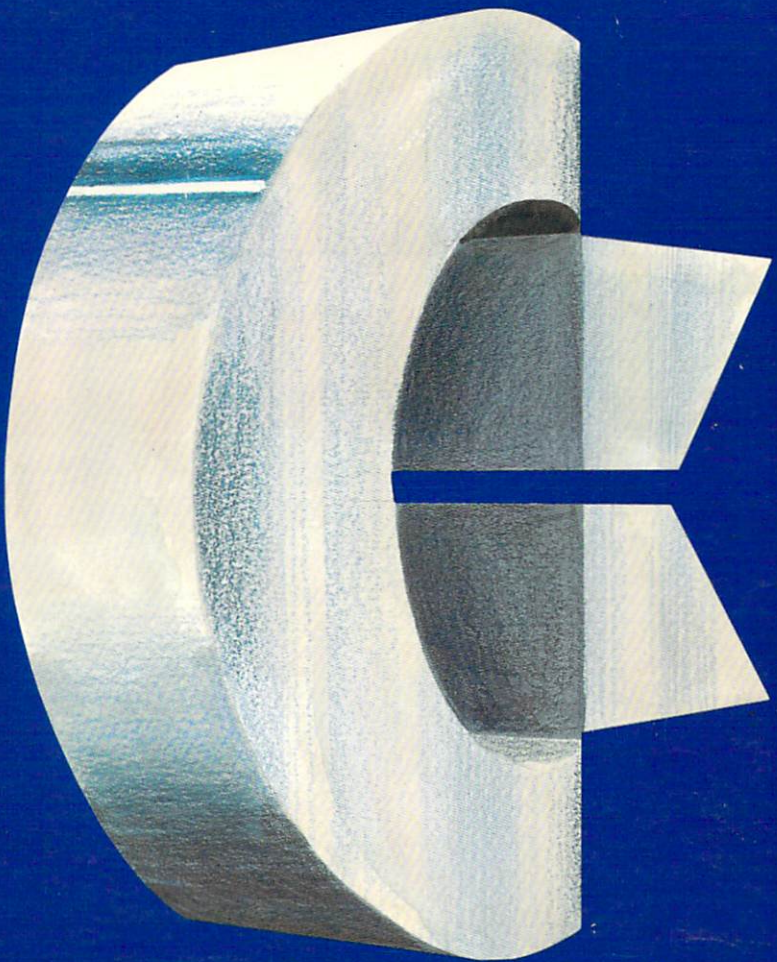
$12.95

COMPUTE!'s FIRST BOOK OF PET™/CBM™



From The Editors of COMPUTE! Magazine

COMPUTE!'s FIRST BOOK OF PET®/CBM®

Published by **COMPUTE!** Books,
A Division of Small System Services, Inc.,
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-01-9

10 9 8 7 6 5 4 3 2

Table of Contents

Introduction

Robert C. Lock, Editor/Publisher, **COMPUTE!** Magazine

In the fall of 1979, **COMPUTE!** Magazine began as a quarterly resource and applications journal for owners and users of a variety of personal computers. Since that beginning, we've been supporting the Commodore PET and the Commodore Business Machine. On the pages that follow, you'll find some of the best of the PET/CBM articles that appeared in **COMPUTE!** during 1980.

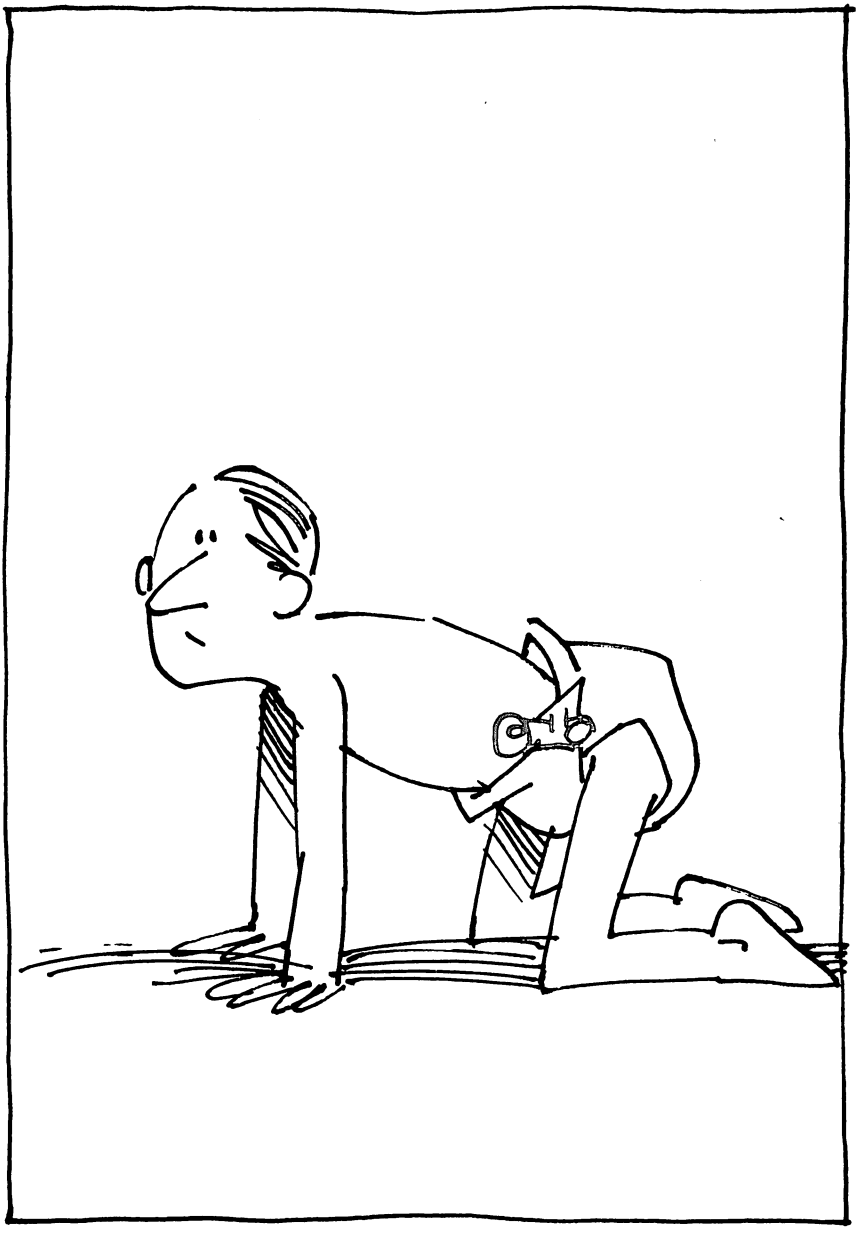
The Commodore product line is continuing to expand, and in the monthly issues of **COMPUTE!** Magazine, you'll continue to find the same commitment and support for the PET, the CBM series, and now, the SuperPet and VIC-20 models. **COMPUTE!**'s internationally recognized authors continue to support, encourage and tutor readers from the beginner to the advanced.

We've organized the following material, and designed the book so that it will be easy to use. If you have any comments or suggestions regarding this book, or future books you'd like to see from us, please let us know.

Our special thanks to Charles Brannon, Richard Mansfield, and Kathleen Martinek of the editorial staff at **COMPUTE!**; Kate Taylor, Dai Rees, and De Potter of the Production staff; Georgia Papadopoulos, Art Director; and Harry Blair, our illustrator.

COMPUTE! Books is a division of Small System Services
publishers of **COMPUTE!** Magazine.
Editorial offices are located at
505 Edwardia Drive, Greensboro, NC 27409 USA. 919-275-9809.

CHAPTER ONE:
Getting Started



A Commodore Perspective

Robert J. Crowell

The history of the Commodore PET computer is a very interesting and mostly unknown story. For the benefit of the vast group of Commodore PET owners, here is a brief history, from the author's experience, on the evolution of the Commodore PET.

The story of the Commodore PET computer began in 1974. MOS Technology, a semiconductor research and manufacturing company in Norristown, PA., was partially purchased by Commodore. This purchase gave Commodore a new technological 'pool' to draw from. In 1975, the founders of MOS Technology, some of whom were from the Motorola 6800 (microprocessor) design group, felt that they could improve on the 6800 microprocessor. The resultant research and development led to the announcement of a whole new series of integrated circuits, the 6500 series. The 6502 CPU "Computer on a chip" microprocessor and future CPU of the PET was born. The announcement of this family of chips did not arouse much excitement except in engineering circles. As the product data sheets on this new family of chips were circulated, engineers spent many hours discussing the future applications of the 6502 CPU, PIAs, VIAs, and other esoteric technological marvels.

At this time, during early 1976, the microcomputer industry consisted of a relatively small number of engineering type hobbyists happily assembling a few microcomputers (the first microcomputers were produced in kit form). Soon various articles began appearing that heralded the availability of microcomputers for everyone.

How many of these "computers on a board" would sell? No one at that time could estimate the market for a technologically new product like microcomputers. As advertisements on this new product continued circulating, the infant "hobbyist" market began clamoring for attention by ordering thousands of these units, mostly on a cash prepaid basis. Send in your funds and wait three to six months for your unit to arrive!

During this period the engineers at MOS Technology, having recently been acquainted with this new industry, announced that they would introduce a new computer on a board called the KIM-1. The KIM-1, with the 6502 as the CPU/brain, became a success before the first unit came off the assembly line. The "father unit" of

the PET was born.

Thousands upon thousands of KIMs were subsequently sold as the hobbyist, industrial, and educational markets adopted the KIM-1 with open arms. Remember, the KIM-1 was designed for easy use. All you had to do was hook up your own power supply, hook up a cassette, and away you went happily programming in hexadecimal format! Even at this early stage it was evident to some people that the KIM's days were numbered. Sooner or later the various markets demanding the KIM-1 would become semi-saturated.

During the introduction of the KIM, a vague shape began forming in the mind of a senior engineer at MOS. Why not design a KIM-like unit that would contain a power supply, an interpretive language (BASIC) to allow non-technical people to program it, a CRT video display, and a keyboard? Could a product like that be sold? Who would buy it? How would the unit be marketed?

As the KIM-1 enjoyed a vast amount of success, MOS Technology was selling integrated circuits for use in calculators and other products. (All you Apple owners can thank MOS Technology for the 6502 Microporcessor!) Enter Commodore in a big way! Commodore, a customer and partial stockholder of MOS, was one of the earliest manufacturers of handheld calculators. MOS Technology had the chip development and manufacturing capabilities to produce chips in large quantities, but did not have a consumer-oriented marketing staff. Commodore did not have any large chip manufacturing capability, but was essentially a marketing firm with offshore calculator manufacturing capabilities. The match was obvious and a very quick "takeover" was arranged. The result was that Commodore became a vertically integrated company, designing and manufacturing chips on one end and selling the finished product on the other end. This vertical integration, in conjunction with the overseas arm of Commodore, laid the foundation that allowed Commodore to announce that an entirely new product was coming. By December of 1976, Commodore's stock had jumped from 4½ to 7.

Who would buy the unit? At what price? What do we call it? As this new unit would probably be sold directly to users in the home, an acceptable name had to be created. Remember, in 1977, very few people had a firm understanding of this new market, and the general consensus was that the lion's share of the market would be people utilizing the unit for "personal" transactions. Since computers were "scary" to the average person (they fill entire rooms

and cost millions of dollars, don't they?), a nice, comfortable product name had to be created. The name Personal Electronic Transactor was quite a mouthful, but, as was originally planned, the acronym P.E.T. became the accepted name.

The original pricing of the unit was announced in mid-1977 at $495 for the 4K RAM unit. The price quickly went to $595 for the 4K unit and a $795 8K (optional) unit was announced.

The industry scoffed and said it couldn't be done at that price. Well, basic marketing philosophy (and good corporate management) dictates that if you come out at the lowest possible price point, with a good possibility of a mass market, you might make small profits (if any) at the beginning, and you make it up in future large volume production. Of course, a low price also helps to preclude market entry from competitors. An ulterior pricing motive may have been to announce a price that would remain stable. In the mid 1970's the pricing in the calculator market continuously decreased as companies "skimmed" the market with one lower price point after another. Due to these regular decreases in price, the purchasing public began waiting for lower prices before they purchased. If the PET was announced at a higher price, say at $1195 and then dropped to $995 and then again to $795 the market would possibly have waited for even lower price points. As the PET, by its very nature, would have a much longer product life cycle than a calculator product, a stable pricing policy became an important consideration.

In June of 1977, Commodore unveiled the PET at the Consumer Electronics Show in Chicago. There was the PET, amidst all the Commodore calculators. The public went wild. I personally stood there and like many others wrote out a check (at full retail) to Commodore to purchase a PET. I watched one person purchase four units. During the three-day Show, Commodore's stock again jumped, this time to 9¼.

Commodore originally announced that the PET was capable of handling many different tasks, especially with their "soon to be available" 2020 printer. Many of these potential uses would, of course, require a printer as well as support from Commodore. However, Commodore never had a chance. After Popular Science put the PET on it's cover the demand for the PET exploded. Commodore quoted 30-day delivery, then 60, then 90, then an astounding 4-5 months! Remember, these were all prepaid orders. Commodore was inundated with customer orders, dealer inquiries, and requests for information. Due to the size of Commodore's staff,

many requests went unanswered, as Commodore concentrated on the task at hand — producing as many PETs as possible.

As Commodore was marketing the PET directly to consumers, the 40 to 50 dealer inquiries received per day piled up. A few persistent dealers continued to clamor for attention.

Due to the absolutely incredible demand for the PET, Commodore was extremely selective of its dealers. Commodore required a service technician, a retail outlet, an excellent credit history, and a cash deposit on future orders. The cash deposit weeded out a large percentage of potential dealers and left Commodore with only financially strong dealers to choose from. A tremendous commitment to the future of the PET and to Commodore was required for a prospective dealer to send a certified check for a large amount of money, with no idea when to expect their deposits back. The required cash deposit also supplied Commodore with short-term working capital, allowing them to maximize production. In early 1978, as demand continued to expand, the Commodore PET dealer network was started.

The dealers who were selected found themselves able to require prepayment from customers: in economic terms, a vertical transfer of funds. Commodore required deposit funds and in turn, the dealer required prepayment; delivery to customers (from dealers) was now 30-60 days. The purchasing public prepaid and prepaid. Commodore's stock rose and rose.

As volume production began in earnest, Commodore (I assume) realized that within a year or so PET production and therefore supply would be close to PET demand. Commodore had increased production, but had not increased their marketing staff to support the large numbers of PETs being delivered. As the PET is a computer, many user and dealer questions arose. Most of these questions went unanswered as the small marketing staff at Commodore was taxed to the limit. In order to expand the markets for the PET, a crash effort began to bring the long-awaited peripheral printer to market.

Problem after problem developed, vendor designs were examined, tested, and discarded one by one. A print head was finally accepted and Commodore announced that the long awaited printer would go into immediate production with deliveries commencing in a few months. After this public announcement, in January of 1978, Commodore found that the print head they had selected did not perform within the specified engineering parameters. The print head mechanism developed problems after

continuous use. An increase in price was announced hoping (I assume) that the extra projected profits would justify a quick re-engineering of the unit. However, this was not to happen. The print head problem, coupled with other problems was enough to force Commodore to cancel the 2020 printer. Back to the drawing board. Within a few months, Commodore announced that they would come out with two new printers, at higher prices, at some point in the future.

Many customers had prepaid 2020 printer orders and the lack of information from Commodore on their orders, coupled with the lack of good documentation on the PET, strained customer and dealer relations. In the midst of all these problems, a larger problem arose. PET production was rising faster than PET demand and soon a production surplus would be created.

A major corporate decision was finally made to bring in some upper echelon personnel to assist Commodore in the transition from the marketing of the 8K PET to the marketing of the CBM business system (large keyboard PETs and peripherals). A secondary charter for the new, upgraded marketing department was to expand the chain of distribution.

The new personnel began to support the Commodore PET line which would soon include the business peripherals: the 2040 Dual Floppy Disk Drive, and the 2023 and 2022 printers. With the new large keyboard PETs, Commodore introduced the long awaited updated version of their operating system and offered a ROM Retrofit Kit to 8K owners. This ROM Retrofit Kit, while eliminating the major problems of the 8K operating system, also allowed 8K owners to utilize the new 2040 Dual Disk. Up until this time, Commodore had only one product to worry about, the 8K PET. However, with the start of production of the new units, suddenly Commodore had eight products to worry about. In February of 1979, concurrent with this new production, Commodore consolidated their operations (previously in three different locations in the Palo Alto, California area) under one roof. Commodore's new 60,000 square foot facility in Santa Clara tremendously simplified production and other logistics.

In April of 1979, all of the new business peripherals became available — suddenly, all the components required for a business system (CPU, Disk Storage, and a printer) were available. Owners and dealers scrambled to understand the 2040 Disk operating system and to write/modify business software to run on this system. Many people in the industry thought that demand would increase

once a full system configuration *and* software were available. However the market didn't even wait for the software and demand increased tremendously. In April of 1979, Commodore announced that they had appointed five regional distributors (an important and necessary move) to provide front end marketing and logistic support to smaller and new dealers. At this time Commodore announced an educational blockbuster. For every two 8K units purchased at retail (from a dealer), Commodore would supply the school with one free 8K unit; an effective discount of 33⅓%! Many people in the industry suggested that this was Commodore's method of cleaning out the existing stock of 8K units. This was not completely true. Since the announcement of the PET, Commodore had not directly advertised the PET. Rather than advertise nationally on a product that was experiencing strong demand, Commodore decided to support the educational market. This program achieved its goal of moving the 8K unit, in large numbers, into the educational markets.

Demand continued unabated on the entire Commodore product line. At the June, 1979 National Computer Conference in New York, Commodore announced a Word Processing program written for the PET system. This excellent program was a milestone in the evolution of the PET system as it was the first viable business-oriented software that would be mass marketed by Commodore. Soon thereafter, various distributors announced a full line of business programs including General Ledger, Mailing List, and the new Word Processing System. For the first time, a fairly complete set of business software was available for the Commodore System. During June of 1979 Commodore's stock, having almost continuously risen, reached an all-time high of 41 5/8!

In my opinion, the Commodore system has the potential of becoming the most popular and widely used small business system of 1980. New products, among them an 8K version with a large keyboard, are being developed. The early 1980's will prove to be an exciting period as the PET system becomes more powerful. If Commodore continues to keep the end consumer in mind and supports their distributor and dealer network, I expect the Commodore product line to continue achieving success. As other manufacturers enter the small computer market, with business systems not far behind, we can expect to see Commodore Marketing maintaining their new emphasis.

This brief history, from my own experience, may not be entirely accurate as far as dates, etc., are concerned. My intent was not to write a technical article. The history of the PET contained

many problem periods; the fact remains that the PET is one of the most powerful and popular microcomputers and was one of the major forces behind the new era of small computers.

My congratulations to Commodore.

ROM-antic thoughts

Jim Butterfield

Should you upgrade to a new ROM set?

Here comes another ROM set or two from Commodore, and once again the user will need to make the decision: should I upgrade?

It's a tough question. It will cost money, and some of your programs may cease to work until they have been modified. If you don't, you'll be left behind and won't have access to some of the new goodies.

BASIC programs will, as always, remain compatible, so long as they don't bristle with obscure PEEKs and POKEs. Machine language itself doesn't change, but programs which use routines built into the ROMs will need changing since the routines will have moved to new locations. Some commercial machine language programs will survive transfer to new ROMs, but many won't.

A more subtle problem creeps in. As the machine is enhanced, programs will start to use the new built-in features, and users may find themselves having to retro-convert these so that they will run on older systems. A command such as DOPEN is convenient and compact, but users who haven't converted up will have to translate this to the appropriate OPEN 1,8,3 . . . command. New disk features will be particularly noticeable for this. New systems, for example, won't need to initialize disk and will offer very simple disk error checking; older systems will need to add extra coding to do these. Some new disk features such as APPEND or Relative files have no counterpart on the old systems.

Some terminology:
Commodore is currently referring to ROM sets by means of a numbering scheme. They translate roughly as follows:

BASIC 1.0 Original ROM, as fitted in the early 4K and 8K PETS. Not good for disk I/O; arrays limited to 256 elements; cassette tape files a little awkward.

BASIC 2.0 Upgrade ROM, fitted on more recent machines. Garbage collection still a problem. Keyboard/disk interface rather clumsy. Built-in machine language monitor. Linefeed output to IEEE a minor problem.

Basic 4.0 New ROM, currently being released. Disk command built into BASIC. Garbage collection fast, and Linefeed problem eliminated. Uses more ROM space. Available for both

40- and 80-column machines, but not for original PET 8K hardware.

BASIC 5.0 Business ROM, not yet released. Rumored to have many BASIC enhancements, including high-precision decimal arithmetic.

BASIC 2.0 and 4.0 have alternate versions for the two types of keyboard — graphics or business.

Disk systems:

DOS 1.0 Original 2040 disk system. INITIALIZE command needed; RENAME sometimes doesn't work.

DOS 2.0 New system, currently being released. INITIALIZE not needed, but allowed. Relative files and APPEND command implemented. Fast BACKUP command. Can be retrofitted to early 2040 units. The new 8050 disk system will have characteristics similar to DOS 2.0

Printer ROM systems haven't settled down yet. There are two systems available, but both have minor problems; a third is rumored.

Upgrading: the Options

Users who still have BASIC 1.0 should upgrade, at least to BASIC 2.0. There are too many good things available.

The original 8K machines cannot be readily upgraded beyond BASIC 2.0; their hardware won't support BASIC 4.0.

It's not necessary to upgrade both BASIC and DOS ROMs at the same time, but it's probably a good idea. BASIC 4.0 and DOS 2.0 work harmoniously together.

Switch to BASIC 4.0 if you need any of the following:

— to be up to date with the latest software;
— to eliminate garbage collection delays;
— to allow inexperienced users to use the disk with more natural, English-like commands.

Switch to DOS 2.0 if you want to take advantage of the new APPEND feature of the powerful relative file structure.

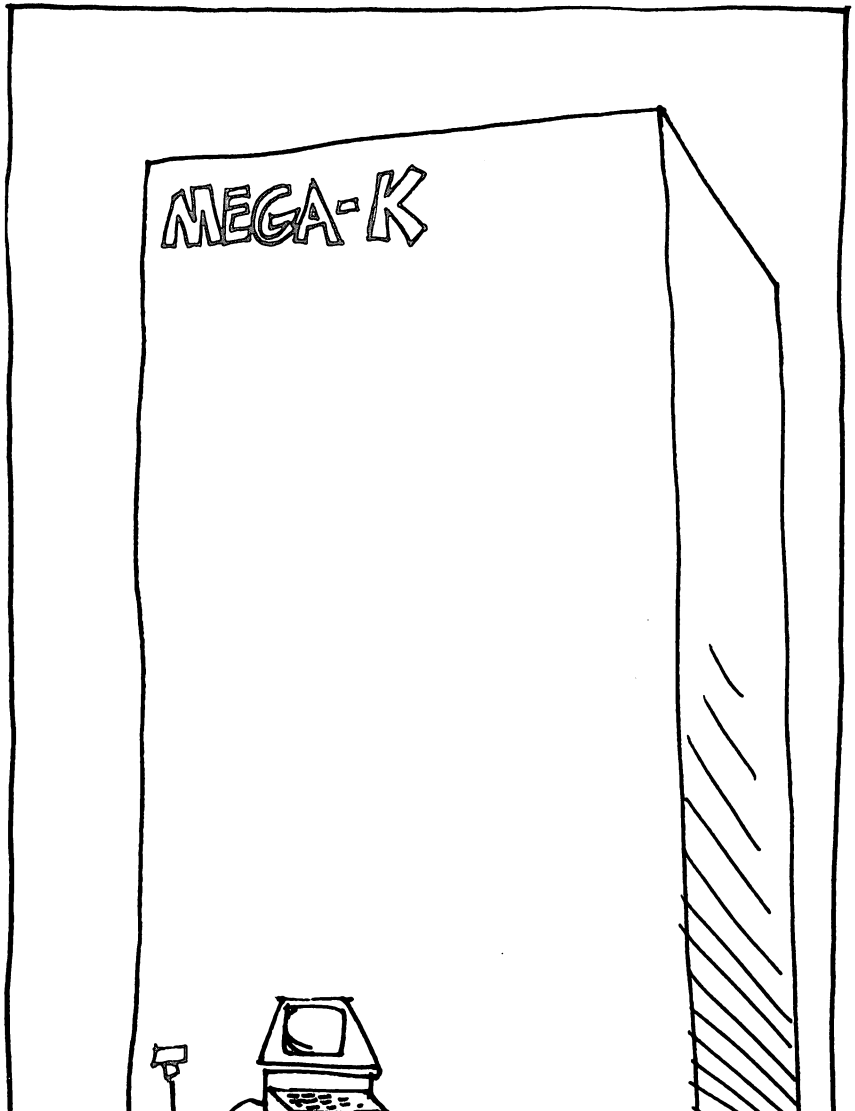
If you still have original ROMs (the ones that say *** COMMODORE BASIC *** upon power-up), plan to upgrade.

It's your option as to whether you want to switch to BASIC 4.0 and DOS 2.0. If you like the new features, go ahead. But you'll still have a good, serviceable system if you stay with upgrade ROM (BASIC 2.0).

Upgrading the disk file can be treated as a separate question.

The original unit is excellent for program SAVEing and LOADing. But if you plan to do a lot of work with data files, the new features of DOS 2.0 can look very attractive.

It's your choice.



Tokens Aren't Just for Subways - A Convenient Method to List Microsoft BASIC Tokens

Harvey B. Herman

The latest buzzword in computer circles is "Tokens." I have even heard the verb "tokenize" used in casual conversation. However, my observation is that many people are still confused about the meaning of this term and would like to learn more. How do you explain to someone looking at the table on p. 8 of the Spring 1979, issue of the PET Gazette (list compiled by Jim Butterfield) why, for example, a decimal 161 in memory can have four or more different meanings, including the three letter BASIC key word GET? This article is intended to clear up some of the confusion (I hope) and to illustrate a convenient method to list all the tokens in various versions of Microsoft BASIC (PET, KIM, SYM, etc.).

Understanding tokens is not just an idle exercise. Useful programs have begun to appear which use "token knowledge" for specific purposes. For example, Len Lindsay recently published (The PET Gazette, Summer, 1979, p. 10) a program to identify PEEK and POKE in BASIC programs so they can be more easily converted to run on PETs with new ROMs. This program searches memory for the PEEK and POKE tokens and would not work unless these values are known. Other Microsoft BASICs have similar, but *not* identical, lists of tokens. To use the Lindsay program on other components it probably would be necessary to change the token values. A BASIC program to list PET tokens is shown and discussed below.

The program can be adapted to other BASICs with only a few changes. Before proceeding to that discussion, a few words about tokens are in order. The concept underlying tokens is not difficult to understand. Programs are not stored exactly as they are typed in. Instead of storing all the characters in the keyword PRINT, for example, PET Microsoft BASIC stores only one 8 bit character, decimal value 153. This saves storage space and speeds up execution of programs. All the tokens are greater than 127, i.e., their

hexadecimal value has its most significant bit (MSB) set. The BASIC interpreter can rapidly identify the tokens by checking the MSB and jumping to the appropriate subroutine.

The number of tokens in a given BASIC depends on the number of commands and functions which have been implemented. In a recent article on tokens (MICRO 15:20) a list for OSI BASIC was included which showed 68 tokens (for comparison PET has 75). Also, the PRINT token had the decimal value of 151 (PET uses 153). These facts are cited to emphasize the importance of modifying programs which PEEK at memory for particular tokens when transferring the programs to other computers. The values may accidentally agree, but don't count on it.

The program shown is LOADed and RUN normally. It converts the REM tokens in statements 128 to 202 (PET version) to the correspondingly numbered token and terminates with a list of the tokens and their decimal and hexadecimal equivalents. Note the program will not run a second time with a simple RUN command as the first REM has been replaced with an END (try RUN 500 instead). The PET version can be listed on a printer, if available, by deleting the REM in statement 500 and properly closing the file after the program ends.

If you are using this program on another computer (KIM or SYM) the number of tokens will need to be changed. The proper value can be found by trial and error. When the number of tokens is less, an error will be printed when the list in statement 550 attempts to print an invalid token. The number of the last printed token is used to correct statement 550. The REM comments will help in locating other statements which use the number of tokens and need correction. When the number of tokens is greater than the PET, more initial REMs should be added (203 and above), and the number of tokens increased appropriately until an invalid token causes an error message as above.

Whatever computer is being used, the list of tokens should be kept handy as it is an invaluable aid in understanding and modifying programs written for other systems.

```
128 REM 80      137 REM 89      170 REM AA      179 REM B3
129 REM 81      138 REM 8A      171 REM AB      180 REM B4
130 REM 82      139 REM 8B      172 REM AC      181 REM B5
131 REM 83      140 REM 8C      173 REM AD      182 REM B6
132 REM 84      141 REM 8D      174 REM AE      183 REM B7
133 REM 85        :             175 REM AF      184 REM B8
134 REM 86        :             176 REM B0      185 REM B9
135 REM 87      168 REM A8      177 REM B1      186 REM BA
136 REM 88      169 REM A9      178 REM B2      187 REM BB
```

```
188 REM BC      192 REM C0      196 REM C4      200 REM C8
189 REM BD      193 REM C1      197 REM C5      201 REM C9
190 REM BE      194 REM C2      198 REM C6      202 REM CA
191 REM BF      195 REM C3      199 REM C7

 500 REM OPEN 5,4:CMD 5
 510 FOR I=1 TO 667 STEP 9:REM 667(9*#TOKENS-8)
 520 J=J+1
 530 POKE 1028+I,127+J:REM 1028(START OF PROGRAM STORAGE+4)
 540 NEXT I
 550 LIST 128-202: REM 202(127+#TOKENS)
 560 REM PRINT#5:CLOSE5
READY.
```

Program (Below) With Output

```
142 REM 8E
143 REM 8F
144 REM 90
145 REM 91
146 REM 92
147 REM 93
148 REM 94
149 REM 95
150 REM 96
151 REM 97
152 REM 98
153 REM 99
154 REM 9A
155 REM 9B
156 REM 9C
157 REM 9D
158 REM 9E
159 REM 9F
160 REM A0
161 REM A1
162 REM A2
163 REM A3
164 REM A4
165 REM A5
166 REM A6
167 REM A7
```

```
128 END 80
129 FOR 81
130 NEXT 82
131 DATA 83
132 INPUT# 84
133 INPUT 85
134 DIM 86
135 READ 87
136 LET 88
137 GOTO 89
138 RUN 8A
139 IF 8B
140 RESTORE 8C
141 GOSUB 8D
142 RETURN 8E
143 REM 8F
144 STOP 90
145 ON 91
146 WAIT 92
147 LOAD 93
148 SAVE 94
149 VERIFY 95
150 DEF 96
151 POKE 97
152 PRINT# 98
153 PRINT 99
154 CONT 9A
155 LIST 9B
156 CLR 9C
157 CMD 9D
158 SYS 9E
159 OPEN 9F
160 CLOSE A0
161 GET A1
162 NEW A2
163 TAB( A3
164 TO A4
165 FN A5
166 SPC( A6
167 THEN A7
```

```
168 NOT A8
169 STEP A9
170 + AA
171 - AB
172 * AC
173 / AD
174 ^ AE
175 AND AF
176 OR B0
177 > B1
178 = B2
179 < B3
180 SGN B4
181 INT B5
182 ABS B6
183 USR B7
184 FRE B8
185 POS B9
186 SQR BA
187 RND BB
188 LOG BC
189 EXP BD
190 COS BE
191 SIN BF
192 TAN C0
193 ATN C1
194 PEEK C2
195 LEN C3
196 STR$ C4
197 VAL C5
198 ASC C6
199 CHR$ C7
200 LEFT$ C8
201 RIGHT$ C9
202 MID$ CA

READY.
```

Big Files On A Small Computer

Elizabeth Deal

The program described here demonstrates a way of reducing data storage requirements by a factor of eight. It is written in Microsoft Basic for a PET computer.

I have seen several programs that create and use cross-index files for library search, statistical surveys, and similar applications. They usually require large computers, such as a 48K system with two disk drives. A very thorough file handling system has been described recently by Dr. Sanger in the November, 1979, issue of *Microcomputing*. In his article, each attribute is coded as two letters and six attributes are permitted for each record. This requires twelve letters and, therefore, twelve bytes.

In the method described here, each attribute is coded as yes or no and the user can have as many attributes as he desires. If the application lends itself to such coding into a list of keys or attributes, then this system will permit the handling of large amounts of data in core at one time. It also permits the use of logical AND, OR or NOT operators in retrieval with any combination of attributes.

By way of illustration, a library search requires quick access to those entries that contain desired subject matter. Two, three, or six byte coding of each key is very core consuming, and limits the number of records that can be in core at one time.

The solution I propose is twofold: (1) set up a smart coding procedure for classification of subjects described in an article into keys that can be scored yes or no, and (2) "pack" the data for storing it in core, on tape or on disk, and then "unpack" it, one record at a time, during the search for the applicable attributes. This paper describes an efficient way to "pack" and "unpack" the data so that a larger file can be searched on a small computer without the use of accessory memory devices, such as disks. Of course, if one has a system with a disk the method described here would permit use of an even larger file. We are aware that the first part of the solution (setting up the coding procedure) is challenging. It is the real problem and the performance of the system depends on how logical and meaningful the selected keys are.

Each logical record consists of the text part and the data part. The text part must be adequate for positive identification of the

articles being searched, but the length should be kept to a minimum. Name, date, and page might be enough. The data part is what we can compress. The yes-no or 1-0 codes are entered in groups of fifteen ones and zeros. These, in turn, are packed into the two byte integer variable S%.

Fifteen attributes require two bytes, thirty attributes require four bytes, and so on. A user of the system need not concern himself with what the program does with binary numbers. He only needs to know that there will be as many S% values per record as there are groups of fifteen keys. The user then needs to provide a decision for retrieving the records of interest to him. The decision is written as a statement at the beginning of a program and is immediately edited for syntax-type errors. Logical operators AND, OR, NOT, as well as arithmetic ones (=,<>,<,>) are used. The decision can be written on one or more lines leading to a combining variable TR. TR is set to one if true, and all records meeting TR condition are then displayed. Complete instructions for writing TR lines are listed in lines 2970 and 3420.

How is it done? For once those long tables of powers of two, that are a part of every book on programming, come in handy. The program is set up in such a way that the user thinks of the list of fifteen keys from left to right, 1 to 15. The program sees them as being numbered from right to left, 0 to 14. Like this:

— Key numbers B%(k)k = 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
— Program sees as m = 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
— Input key values 1 0 0 0 1 1 1 1 1 1 0 0 0 0 0

The program now takes key values and wherever it finds a "1" it raises 2 to the m-th power. The sum of all this is then stored in integer variable S% (record number, sum number)*. The bytes are used instead of at least 15. During the process of retrieval the opposite procedure takes place — the sum is "unpacked" into working storage of 15 values. The same values are, of course, reused by all records. The lines of the program that drive this system are 1470 to 1510 and 1920 to 1990 the other way. It seems like a lot of hassle, but the core saving is tremendous. The loops that do the packing and unpacking take from 0.2 second to 0.9 second, the latter representing all fifteen bits on. (These times could be reduced by rewriting these two loops as machine code subroutines.) Another way to save time would be to set up the most frequently used keys next to one another as this will leave the loop sooner. In the example shown above, the program will loop ten times. Had a "1" been in position 4 or 11 the loop would be

executed five times.

The program now has two sections. One packs the data, the other unpacks it. In between, the values should be stored on tape. And at the beginning, routines for creating and updating files should be provided. As listed, the program works as if it were a file system. It can be used as a training ground in writing decision lines. It should be used as a part of a larger system.

To customize the program for your system (1) delete lines 2770 on to reduce core from 10K to 5.2K, (2) in line 1020 insert the maximum number of records that will fit, (3) modify G if you wish in lines 1020 and 1340 (G = 1 or 2 is now permitted), and (4) insert your machine size in line 2450.

As written, the program takes about 5.2K after removal of all REM lines that are at the end. A search type program that would contain a dictionary of keys should take no more than 6K. How many records can various systems handle? If we assume that each record has 26 bytes of text plus 30 yes-no attributes, an 8K system could search 66 records, a 16K system could search 330 records, and a 32K system could search 866 records. A 100K disk would add 3,333 records and a half-megabyte disk would add 16,500 records.

* $S\%(nr,g) = 2^5+2^6+2^7+2^8+2^{10}+2^{14} = 18{,}400$

Credits: Ken Brossoie

Microcomputing, November, 1979, page 44.

Neil Harris, A-B Computers, Montgomeryville, Pa.

```
1000 C0=0:C1=1:C2=2:C3=3:C5=5:C6=6:
      ¬CA=10:CE=14:CF=15:D$=""
1010 REM ELIZABETH DEAL,MALVERN,PA
1020 R= 200:G=2:DIM HD$(R),S%(R,G),
      ¬WA$(15),V$(G),B%(15*G),QQ%(15)
1030 FORJ=C1TOCF:READWA$(J):NEXTJ
1040 FORK=0TOCE:QQ%(K)=C2^K:NEXT
1050 :
1060 :
1070 REM RENUMBER 1000,10 WHEN NEEDED
1080 :
1090 REM>>>>  R E A D   T H I S  <<<<
1100 REM        INSERT DECISION
1110 REM  KEEP LST-TR-RTRN SEQUENCE
1120 :
1130 LIST-1280
1140 DX=ABS(B%(1)ORB%(2)):
1141 DY=ABS(B%(3)ORB%(4)):
1142 DZ=ABS(NOT(B%(5)ANDB%(6))):
```

```
1143 TR=ABS(DXANDDYANDDZ):
1150 RETURN
1160 :
1170 REM>>>>> ':' MUST END EACH LINE
1180 :
1210 REM TR=ABS((B%(1)ANDNOTB%(2)) //
1220 REM TR=ABS(S%(N,2)>512)WILL DIS-
1230 REM PLAY RECS MEETING STMT TR.
1240 :
1250 REM >> TYPE 'RUN', VERIFY, THEN
1255 REM    TYPE 'GOTO1300' TO CONT.
1260 REM      (OR ~ ~  THIS LINE#)
1270 :
1280 REM ========= SECTION 1 =========
1290 :
1300 A=TI:REM //CHOSE SEC.1/2 HERE ///
1310 GOSUB2420:REM     // SUBR 1   ///
1320 PRINT"ñ":N=1:LT=0
1330 PRINT"HOW MANY GROUPS OF 15 KEYS ¬
     ¬?":PRINTTAB(9)"ENTER 1 OR 2"
1340 INPUTG:PRINT:IFG<1ORG>2GOTO1330
1350 PRINT:PRINT:PRINT"ENTER TEXT,
     ¬ OR 'XX' TO END INPUT"
1360 PRINTTAB(2)"!";TAB(27)"!"
1370 INPUTHD$(N):IFLEFT$(HD$(N),
     ¬2)="XX"THEN1550
1380 HL=LEN(HD$(N)):E=0
1390 REM // INPUT15 BITS,FLAG ERRORS
1400 :FORJ=C1TOG:PRINT:PRINT"15 KEYS";
1410 PRINT"  !!!!!!!!!!!!!!!" :PRINTTAB(
     ¬7);:INPUTV$(J)
1420 :FORLL=C1TOCF:S$=MID$(V$(J),LL,1)
1430 IF(S$<>"0"ANDS$<>"1")THENE=E+1:
     ¬GOTO1450
1440 :NEXT
1450 :NEXTJ
1460 IFE>0THENPRINT:PRINT"rERROR DO ¬
     ¬AGAIN":GOTO1350
1470 TX=TI :FORJ=C1TOG:S%(N,J)=C0
1480 :FORL=C0TOCE:PQ=VAL(MID$(V$(J),
     ¬CF-L,C1)):IFPQ=OTHEN1500
1490 S%(N,J)=S%(N,J)+QQ%(L)
1500 :NEXTL:PRINTTAB(14)"SUM="S%(N,J)
1510 :NEXTJ:TY=TI
1520 PRINT"(";INT((TY-TX)/C6)/CA;"SEC)";
     ¬:PRINTTAB(25)"OK";N;"OF";R
1530 N=N+C1:LT=LT+HL:IFN<=RGOTO1350
```

```
1540 PRINT:PRINT:PRINTTAB(6):PRINT"*****
     ¬** NO MORE ROOM *******"
1550 PRINT:PRINT:PRINT:PRINT"# OF ¬
     ¬RECORDS PUT IN";N-1:PRINT
1560 PRINT"# OF BYTES USED BY TEXT";LT
1570 PRINT"# OF BYTES USED BY KEYS";
     ¬2*G*(N-C1):PRINT:PRINT"BYTES ¬
     ¬LEFT";FRE(0)
1580 PRINT:PRINT:PRINT
1590 PRINT"HIT 'S' TO STOP":PRINT"ANY ¬
     ¬KEY TO CONTINUE"
1600 GETA$:IFA$=""THEN1600
1610 IF A$="S"THENSTOP: REM/CHANGE/
1620 :
1630 REM // STORE DATA ON TAPE HERE
1640 :
1650 REM ======= SECTION 2 ==========
1660 REM
1670 REM >> # OF RECORDS (NR), TEXT
1680 REM  REC (HD$(NR)) AND  G-SUMS
1690 REM  S%(R,G) ARE USED IN THIS
1700 REM  SECTION; BITS ARE COMPUTED
1710 REM  AND ASSIGNED TO KEYS ARRAY
1720 REM           B%(G*15)
1730 REM >> B% OR S%  ARE CHECKED FOR
1740 REM  COMPLIANCE WITH TR STMT.
1830 REM
1840 REM ============================
1850 REM
1860 PRINT"ñ":NR=N-1
1880 :FORN=C1TONR:K=C0:JA=TI
1890 IFKS>C0THEN2000
1910 FORK=C1TOG*CF:B%(K)=0:NEXT
1920 JS=TI:FORJ=C1TOG:TP=S%(N,J):
     ¬JJ= J-C1
1930 IFTP=C0THEN1990
1940 Q=INT(LOG(TP)/LOG(C2)):U=CF-Q
1950 :FORM=C0TOQ:BP=QQ%(Q-M)
1960 IF(TP)>=BPTHENB%(CF*JJ+U+M)=C1:
     ¬TP=TP-BP
1970 IFTP=C0THEN1990
1980 :NEXTM
1990 :NEXTJ:JE=TI
2000 FORM=1TO4:PRINT"==========";:NEXT
2005 PRINT:PRINTHD$(N)
2010 GOSUB1140:IFTR<>1THENPRINT:
     ¬PRINT"*** NO MATCH ***":GOTO2030
```

```
2020 PRINT:PRINT"THIS RECORD MATCHES:"
2030 PRINT:PRINTTR$
2040 :FORJ=C1TOG:
2050 PRINT"GROUP";J;"SUM=";S%(N,J):
      ¬NEXTJ:PRINT
2070 IFKS>C0THEN2130
2080 K=C1:PRINT:FORJ=C1TOG
2090 PRINT"BIT";(J-C1)*(K-C1)+C1;:
      ¬PRINTTAB(7);" -> ";:PRINTTAB(12);
2100 :FORK=C1TOCF:PRINTRIGHT$(STR$(B%(CF
      ¬*(J-C1)+K)),1);
2110 :NEXTK:PRINT"  <- ";J*(K-C1)
2120 :NEXTJ
2130 PRINT:PRINT:PRINT"(TIME IN BIT ¬
      ¬LOOP";INT((JE-JS)/6)/10;"SEC)"
2140 PRINT"(TOTAL TIME:";INT((TI-JA)/6)/
      ¬10;"SEC)"
2150 :FORM=1TOCA:PRINT"====";:NEXT
2160 PRINT:PRINTTAB(11)"HIT ANY KEY FOR ¬
      ¬MORE"
2170 PRINTTAB(6)"'Q' TO QUIT AND RERUN ¬
      ¬OPTIONS"
2180 GET A$:IFA$=""GOTO2180
2190 IFA$="Q"GOTO2210
2200 :NEXTN :REM //  END REC LOOP  /
2210 PRINT"ñ↓↓"TAB(16)"OPTIONS":PRINT
2220 PRINT"1. SAME TR, NEW DATA ¬
      ¬STARTING AT REC";NR+C1
2230 PRINT"2. RERUN: SAME TR AND DATA"
2240 PRINT"3. Q U I T"
2250 GETD$:IFD$<"1"ORD$>"3"GOTO2250
2260 ONVAL(D$)GOTO2270,1880,2280
2270 N=NR+C1:GOTO1350
2280 PRINT"↓↓↓SURE ?"
2290 GETA$:IFA$="N"GOTO2210
2300 IFA$="Y"THEN END: REM /PROG END/
2310 GOTO2290
2320 REM  === DATA=FOR SUBR.1 =======
2330 :
2340 DATA"NOT","","+","-","*","/","",
      ¬"AND"
2350 DATA"OR",">","=","<","","","ABS"
2360 :
2370 REM ============================
2380 :
2390 REM         SUBROUTINE 1
2400 REM FIND AND EDIT TR STATEMENT
```

```
2410 :
2420 TR$="":LX=0:KB=0:KS=0:P=0:J1=1:
      ¬KC=0:L1=0:R1=0:SB=0:JA=0:JZ=0
2440 PRINT"ñFOUND 'LIST' AT";
2450 :FORJ=1350TO32000-FRE(0):IFPEEK(J)=
      ¬155THENPRINT"   ***";J: JA=J
2460 IFPEEK(J)=142THENJZ=J:PRINT"FOUND ¬
      ¬'RETURN' AT ***";JZ:GOTO2475
2470 :NEXTJ
2475 IFJA=0ORJZ=0THENPRINT"CAN'T FIND ¬
      ¬DECISIONS; SEE SUB1 LISTING ":STOP
2480 :FORJJ=JA+11TOJZ:LL=PEEK(JJ)
2490 IFLL=58THENJJ=JJ+5:LP$=CHR$(13):
      ¬GOTO2540
2500 IFLL<128THENLP$=CHR$(LL):GOTO2540
2510 :FORM=168TO182:IFLL=MTHENLP$=WA$(M-
      ¬167):LX=LX+LEN(LP$):GOTO2530
2520 :NEXTM
2530 IFM=182THENSB=SB+1
2540 LX=LX+1
2550 IFLX>255THENPRINT"STRING TOO ¬
      ¬LONG-LIMIT 255":E5=1:GOTO2700
2560 TR$=TR$+LP$
2570 :NEXTJJ
2580 :FORJJ=J1TOLX+1
2590 M$=MID$(TR$,JJ,1):KB=KB-(M$="B"):
      ¬KS=KS-(M$="S"):P=P-(M$="%")
2600 KC=KC-(M$=","):L1=L1-(M$="("):
      ¬R1=R1-(M$=")")
2610 NEXTJJ
2620 PRINT" B ";" S ";" % ";" , ";" ( ";
      ¬" ) ";"ABS";" L "
2630 KB=KB-SB:KS=KS-SB
2640 PRINTKB;KS;P;KC;L1;R1;SB;LX
2650 PRINT:PRINTTAB(5)"YOUR DECISION IS:
      ¬":PRINT:PRINTTR$:PRINT
2660 E1=(KB>0)AND(KB<>P):E2=(KS>0)AND(KS
      ¬<>P):E3=(KS>0)AND(KS<>KC):
      ¬E4=(L1<>R1)
2670 IFE1ORE2THENPRINT"* USE % -(B% OR ¬
      ¬S%) !!":PRINT"  USE NO OTHER B, S,
      ¬ %"
2680 IFE3THENPRINT"* USE S%(N,#)FOR ¬
      ¬2-DIM ARRAY"
2690 IFE4THENPRINT"* ( ) DON'T MATCH:
      ¬";L1;"LEFT, AND";R1;"RIGHT"
2700 IFE1ORE2ORE3ORE4ORE5THENPRINT:
```

```
      ¬PRINT"TYPE 'RUN' TO FIX":STOP
2710 PRINTTAB(5)"HIT ANY KEY TO ¬
      ¬CONTINUE"
2720 PRINTTAB(3)"OR 'STOP' TO CORRECT/CH
      ¬ANGE"
2730 PRINT:PRINTTAB(5)"THEN TYPE  'RUN' ¬
      ¬ TO FIX"
2740 PRINT:PRINT"(EDITING AND RE-RUN ¬
      ¬WIPE OUT DATA)"
2750 GETA$:IFA$=""THEN2750
2760 RETURN
2770 REM ============================
2780 REM EXAMPLE OF INPUT FOR ONE
2790 REM LOGICAL RECORD WITH TEXT
2800 REM AND 2 GROUPS OF 15 ATTRI-
2810 REM BUTES EACH, STORED AS
2820 REM 25+4 BYTES.
2830 REM
2840 REM >TEXT:
2850 REM    !                        !
2860 REM    1.MAG.NAME/11-78/  /P.106
2870 REM >FIRST 15KEYS
2880 REM    111010000001011
2890 REM >SECOND 15KEYS
2900 REM    001001000100010
2910 REM >END OF DATA
2920 REM    !                        !
2930 REM    XXXX
2940 REM
2950 REM ============================
2960 REM
2970 REM POSSIBLE USES OF TR LINE(S)
2980 REM
2990 REM 1.DECISION IS WRITTEN AT THE
3000 REM START OF A RUN AND CANNOT BE
3001 REM CHANGED DURING THE RUN.
3005 REM
3010 REM 2.QUICK LISTING OF RECORDS
3020 REM WHICH MIGHT HAVE ANY KEYS
3030 REM ON WITHIN SOME GROUP --
3040 REM TR=ABS(S%(N,#)>=512 AND
3050 REM         S%(N,#)<=4096):
3060 REM WILL DISPLAY TEXT OF RE-
3070 REM CORDS THAT HAVE SOME BITS
3080 REM FROM 3 TO 6 ON (15-12,15-9)
3085 REM
3090 REM 3.SLOWER LISTING OF RECORDS
```

```
3100 REM       (TEXT AND BITS):
3105 REM
3110 REM 3.1 ALL RECORDS
3120 REM TR=ABS(B%(1)ORB%(1)=0):
3125 REM
3130 REM 3.2 SOME RECORDS - USE AND
3140 REM ,OR,NOT AS YOU DO IN LOGIC
3150 REM BUT RETAIN 'ABS' IN ALL
3160 REM EXPRESSIONS. END EACH LINE
3170 REM WITH A ':'. SIMPLE EXPRES-
3180 REM SIONS WILL SHOW UP VIA
3190 REM SUB1. FOR-LOOPS AND OTHER
3200 REM TOKENS WILL NOT,BUT THE
3210 REM PROGRAM WILL STILL RUN.
3240 REM
3250 REM 4.USE ( ) CAREFULLY -
3260 REM NOT(XANDZ)<>NOTXANDNOTZ
3280 REM WORXANDYORZ<>(WORX)AND(YORZ)
3290 REM
3300 REM 5.AVOID VARIABLES WITH B,S,
3310 REM % IN THEM - ONLY ABS AND
3320 REM NAMES LISTED ABOVE ARE LEGAL
3330 REM
3340 REM 6.DECISION LINES ARE SCANNED
3350 REM FIRST FOR OBVIOUS ERRORS, SO
3360 REM WE DON'T LOOSE DATA LATER
3370 REM BY EDITING THESE LINES.
3380 REM
3390 REM 7.LOGIC OF PROGRAM DEPENDS
3400 REM ON USE OF ABS FUNCTION -
3410 REM USE FORMAT: 'VAR=ABS(  ):'
3420 REM ON ALL LINES.
3430 REM ============================
3440 REM
3450 REM PURPOSE : FOR ANY INFORMA-
3460 REM TION PROCESSING WHERE DATA
3470 REM CAN BE CODED INTO Y/N FORM.
3480 REM
3490 REM STORAGE (CORE/TAPE/DISK)
3500 REM REQUIREMENTS ARE REDUCED BY
3510 REM AT LEAST A FACTOR OF 8 AS
3520 REM COMPARED TO CONVENTIONAL
3530 REM CODING OF KEYS FOR SEARCH.
3540 REM
3550 REM THIS SYSTEM WILL STORE 15
3560 REM KEYS-ATTRIBUTES-Y/N THINGS
3570 REM IN BASIC'S 2-BYTE INTEGER
```

```
3580 REM VARIABLE.
3590 REM
3600 REM TEXT SHOULD BE LIMITED TO
3610 REM MINIMUM THAT WILL POSITI-
3620 REM VELY IDENTIFY A RECORD.
3630 REM
3640 REM ============================
3650 REM
3660 REM ELIZABETH DEAL, MALVERN,PA
```

PETting With A Joystick

Harvey B. Herman

Here is the necessary background and hookup information for adding a joystick to your PET. See also "Joystick Revised" for a simple programming interface.

My older style PET keyboard gets banged around quite a bit when my kids play games which use the number pad. A recent PET-Pourri column in Kilobaud (1) prompted me to install a joystick on my PET in order to save my keytops from further wear. This article is intended to share my experiences with this project and to encourage other fumblethumbs like myself to try it.

I purchased the Atari joysticks from Sears as suggested in the above column (catalog #6C99835) for $9.95 each. Since I could not find a mating connector, I cut off each end and attached them to a User Port connector as per the instructions (see also refs 2 and 3). The latter connector can be purchased from any number of companies (e.g., AB Computers). Four signal diodes (1N662), whose specifications I believe are not critical, are also used in this super simple interface circuit which can be constructed in about ½ hour. Check to make sure that the diode cathodes are connected as shown in the circuit diagram (1) and the color coded wires of the joystick are connected to the proper pins of the User Port connector. Otherwise no special precautions are necessary. I did it right the first time (yes, brag!)

This neat hardware would, of course, be useless without software to work it. *Cursor* magazine (4) has supplied several programs which have a joystick option. I tried these first (Demon, Canyon, Pickup and Nab) with success. These programs are written to work with various model joysticks wired and oriented in different ways. Since my configuration was fixed, I modified the joystick subroutine in each *Cursor* program to skip the test step. That procedure can be tedious if a program is run repeatedly. The following changes in the *Cursor* joystick subroutine should work for all Atari-type joysticks wired according to the circuit diagram in reference 1:

```
61030 PRINT: FOR I=0 to 5: READ T, P:
    GOSUB 61120:
    T (I)=T:J(T)=P: NEXT I
61120 T=INT (T/16) AND T: RETURN
61250 DATA 255, 5, 223, 4, 239, 6 127, 8 191, 2, 63, 0.7, 1, 9, 3
```

I ran the original program once to find the values in the T array and used DATA statements in the modified program in order to skip the test step. This considerably speeds up the beginning of a game. I have deliberately ignored the rationale behind the bit manipulations in statement 61120. It is not necessary to understand that in order to use joysticks. I emphasize this point because I hope it will encourage PET users who may strain at bits to attempt projects such as described here. In a future article I may try my hand at a tutorial for those who wish to delve into this mystery further.

If a program was not written with a joystick in mind another modification procedure must be used to convert it away from number pad use. As an example, I modified the program Obstacle (5) which utilizes the full keyboard as two pseudo-joysticks. Each player manipulates his piece (screen character) with the new standard keyboard patterns. "W, X, A and D" and "8, 2, 4, 6". The object is to keep from running into the screen trail left by the other player — the first to do so loses. As with many games, it is easier to use than to describe. The following statements in the original program are used to sense a keypress by a player and change direction if necessary:

```
260 GET R$
265 IF R$= "W" THEN AD=1
          ●
          ●
          ●
300 IF R$= "8" THEN BD=1
          ●
          ●
          ●
```

If W is pressed, the direction of the left player's piece is changed to up. If 8 is pressed, the direction of the right player's piece is also changed to up. Player and direction are determined by the above keyboard pattern which can be learned quickly by new players.

Converting a program like this to joystick use is very easy. The following statements will do this:

```
260 M=PEEK (59471)
265 IF (M OR 240) = 247 THEN AD=1
          ●
          ●
          ●
300 IF (M OR 15) = 127 THEN BD=1
          ●
          ●
```

The elipsis can be fleshed out with the help of the table below. A peek at the User Port gives a unique value for each position of the joystick as long as only one is being used. It is necessary to "mask" with 15 or 240 if the possibility exists of both being used at the same time. If only one joystick is used and it doesn't matter which, a further operation with the PEEKed value generates simpler numbers (shown in the last column of the table). This "trick" was used in statement 61120 in the cursor subroutine and the resulting values used in the main body of the program.

I feel this is an excellent project for a beginner. It is not difficult to modify existing programs for joystick use. Original PET owners with disappearing keytops will appreciate the saving on wear and tear. To be fair, I should say that every program is not suitable for conversion. When fine control of movement is required, joysticks may be difficult to use. Some players with poor hand-eye coordination may still prefer the keyboard. As for me, it seems quite natural to chase and catch some seemingly elusive demon with my movements under reflex control by a joystick.

REFERENCES

1. **Kilobaud Microcomputing,** Robert W. Baker, January 1980, p. 14
2. **PET User's Group Newsletter,** Vol. 0, No. 3, p. 6, 1978
3. **Best of PET Gazette,** Chuck Johnston, p.42, 1979
4. **Cursor Magazine,** P.O. Box 550, Goleta, CA 93017
5. **Dilithium Press,** P.O. Box 92, Forest Grove, OR 97116

TABLE

T=PEEK (59471)

POSITION	Joystick 1	Joystick 2	T= INT (T/16) AND T
center	255	255	15
left	223	253	13
right	239	254	14
up	127	247	7
down	191	251	11
button	63	243	3
left up	95	245	5
right up	111	246	6
left down	159	249	9
right down	175	250	10
mask	T OR 15	T OR 240	

Joystick Revised

Harvey B. Herman

Joysticks add a new dimension to games. Moving a stick to indicate direction is so much more natural than pressing a key.

My previous article, "PETing with a Joystick," (Compute #4) gave some general hints on how to interface a joystick to the PET. In particular, I showed how to modify BASIC programs so that they worked with this device. Nevertheless, it is still a pain to change the large number of old programs that may already be in one's library. I looked around for an easier way to interace with the joysticks that would entail minimal modification of pre-existing programs. This article discusses a machine language program that allows one to use joysticks, without any changes in programs which use the conventional keyboard pattern to indicate the direction of player movement.

Last year *Kilobaud Microcomputing* (March, 1979) had an article on the PET User Port. The author, Gregory Yob, illustrated some software which would service an additional keyboard connected to the User Port. The PET interrupt routine uses a pointer in RAM which can be changed to point to special User code. In the PET, every 60th of a second an interrupt is generated to service, among other things, peripherals like the screen and keyboard. It is possible to modify the pointer to service a foreign device (in the above article the new keyboard) before proceeding with the normal interrupt processing. I used this idea to write a machine language program which would service a joystick connected to the User Port.

The joystick controller program (shown in the figure) has three entry points. The first (XON) is used to manually initialize the interrupt vectors, program variables and User Port. The second (XOFF) is used to manually disable the operation of the program and to restore the old software vectors. The third (PCODE) is automatically entered after the next interrupt once the program has been initialized.

The code from line 500 on is the main section of the joystick handler. It puts a number in the keyboard buffer which indicates the direction to move next. This number, of course, depends on the position of the joystick. The program attempts to minimize contact bounce by determining if the character is stable at the User Port for two successive (2 x 1/60 sec) jiffies. This value can be experimented with to suit one's taste. The program follows the Yob example and

does not allow the keyboard buffer to overflow. A simplified flowchart of the main section is shown in the figure. Modifications necessary for new ROMs are shown in the table (not tested).

The number returned from the User Port falls in the range 63 to 255 (decimal). It is converted by a shift and AND operation to a unique number in the simpler scale of 3 to 15 (decimal) (INT(T/16) and T). This number in turn, is converted by table look up to an indicator of direction movement and stored in the keyboard buffer. A simple example should help to clarify matters. Use the table in my previous article for reference.

If joystick 1 is positioned left, the hex value read at the User Port is DF (223 decimal). The binary equivalent of this number is:

1101 1111

Next shift right four times (INT(T/16)) and the number is:

0000 1101

Finally, AND with the original number (AND T).

The result of these operations:

0000 1101

is 0D hex or 13 decimal.

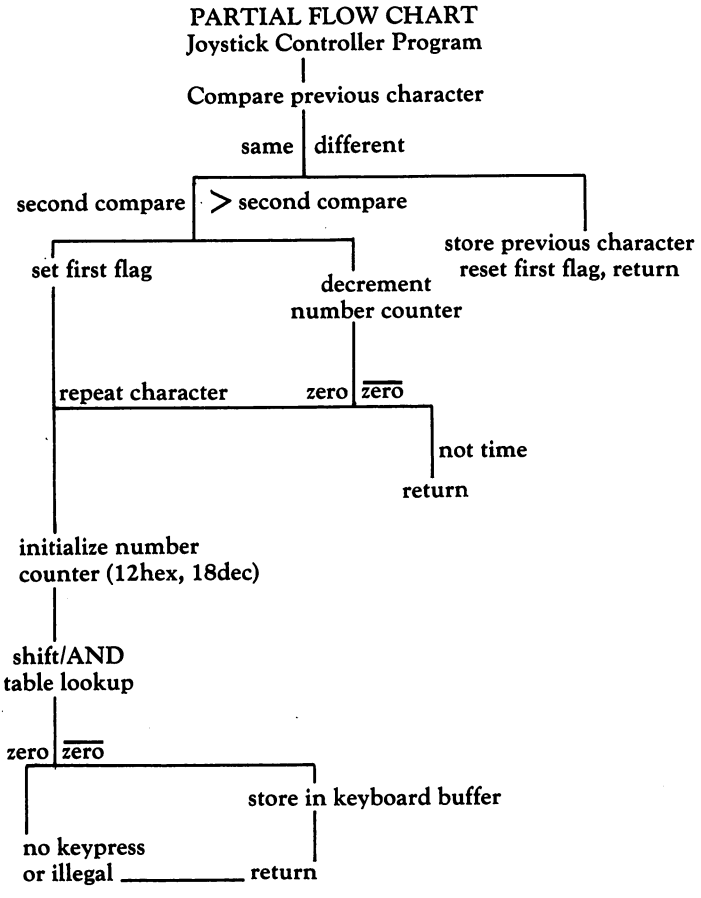
This number is *not* in the normal keypad movement pattern. However, we can convert it to "move left" by reading a table whose 14th value is ASCII 4. The latter number is stuffed into the keyboard buffer. By positioning the joystick left, we end up with the same result as if we had typed 4 on the keyboard. BASIC programs which use 4 to move left can be used without *any* modifications. A similar analysis applies to both joysticks and to all other numbers in the normal keypad movement pattern. In addition, I chose to make the button press mean the number five and straight up mean no keypress.

The hex values given in the listing, can be converted to data statements using a program which appeared in the PET Gazette (H. Sherman, Spring '79, p. 14). Alternatively, one could use a monitor program to load this program. In either case a monitor program is probably the best way to enter the hex values initially.

The joystick controller program described here has been helpful to me when converting programs that use one joystick. A slightly more complicated version would be necessary if one needs to use both joysticks in a two player game. I will be happy to answer questions about the interface or program if you include a SASE.

TABLE
Old PET/New PET Equivalent Locations

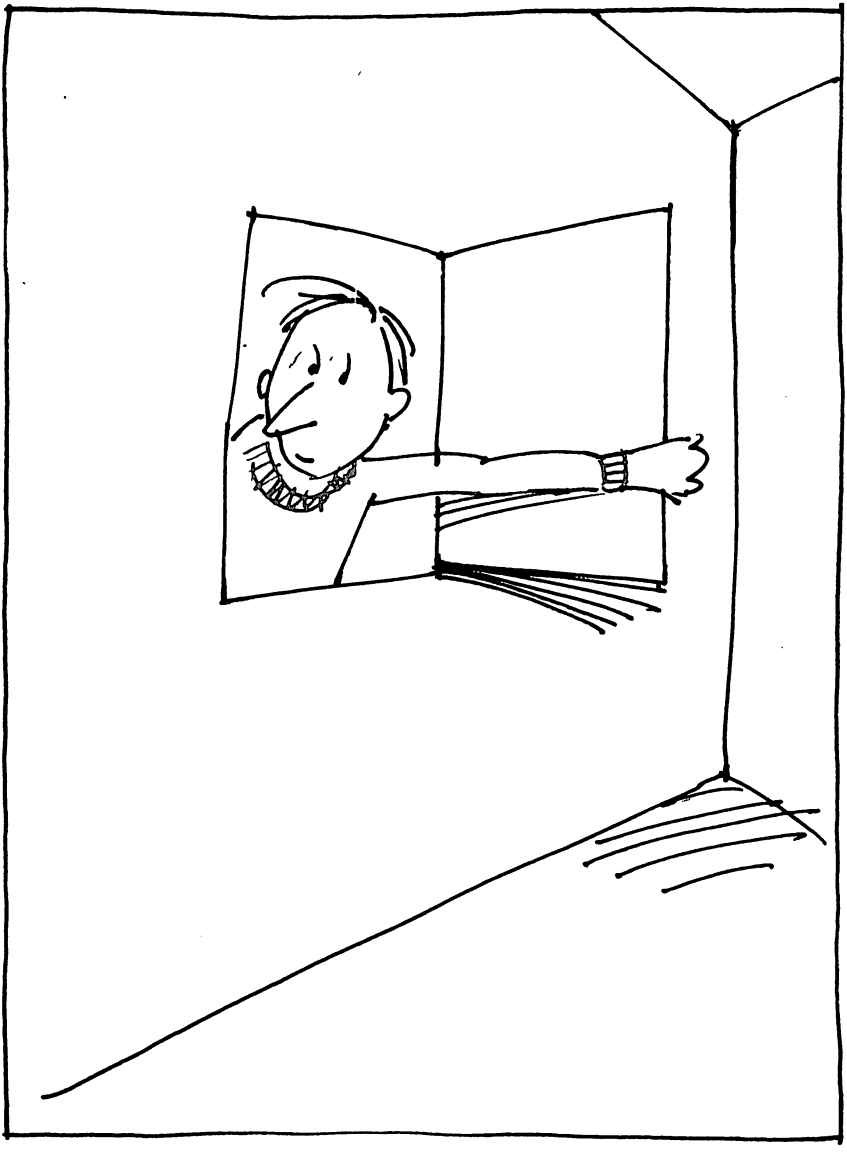
Description	Old	New
Hardware interrupt vectors	$219/$21A	$90/$91
Hardware interrupt routine	$E685	$E62E
Keyboard buffer index	$20D	$9E
Start of keyboard buffer	$20F	$26F
Interrupt return	$E67E	$E6E4



```
                    0110 ; JOYSTICK CONTROLLER PROGRAM
                    0120          .BA $33A
                    0130 ; DISABLE INTERRUPTS
033A- 78            0140 XON      SEI
                    0150 ; SET UP NEW INTERRUPT VECTOR
033B- A9 6A         0160          LDA #$6A
033D- 8D 19 02      0170          STA $219
0340- A9 03         0180          LDA #$3
0342- 8D 1A 02      0190          STA $21A
                    0200 ; INITIALIZE USER PORT
0345- A9 00         0210          LDA #$0
0347- 8D 43 E8      0220          STA $E843
                    0230 ; INITIALIZE PROGRAM VARIABLES
034A- A9 FF         0240          LDA #$FF
034C- 8D CB 03      0250          STA PREV
034F- 8D CC 03      0260          STA FIRST
                    0270 ;ENABLE INTERRUPTS AND RETURN
0352- 58            0280          CLI
0353- 60            0290          RTS
                    0300 ; DISABLE INTERRUPTS
0354- 78            0310 XOFF     SEI
                    0320 ; RESTORE INTERRUPT VECTOR
0355- A9 85         0330          LDA #$85
0357- 8D 19 02      0340          STA $219
035A- A9 E6         0350          LDA #$E6
035C- 8D 1A 02      0360          STA $21A
                    0370 ; ENABLE INTERRUPTS AND RETURN
035F- 58            0380          CLI
0360- 60            0390          RTS
                    0400 ; ADJUST STACK
0361- A9 00         0410 STAX     LDA #$0
0363- 48            0420          PHA
0364- 48            0430          PHA
0365- 48            0440          PHA
0366- 48            0450          PHA
                    0460 ; CONTINUE INTERRUPT PROCESSING
0367- 4C 85 E6      0470          JMP $E685
                    0480 ; MAIN JOYSTICK HANDLER
                    0490 ; LOOK AT USER PORT
036A- AD 4F E8      0500 PCODE    LDA $E84F
                    0510 ; SAME?-YES.
                           SECOND?-ADD TO KEYBOARD BUFFER.
                    0520 ;      THIRD OR GREATER?-KEEP COUNT.
                    0530 ;            ADD TO BUFFER IF > 12 TIMES
                    0540 ;            RETURN IF NOT
                    0550 ; NO.  RESTORE VARIABLES AND RETURN.
036D- CD CB 03      0560          CMP PREV
0370- F0 0B         0570          BEQ EQ
                    0580 ; NOT SAME
0372- 8D CB 03      0590          STA PREV
0375- A9 00         0600          LDA #$0
0377- 8D CC 03      0610          STA FIRST
037A- 4C B4 03      0620          JMP FINISH
                    0630 ; SAME
037D- 2C CC 03      0640 EQ       BIT FIRST
0380- 10 08         0650          BPL SKIP
                    0660 ; THIRD OR GREATER
0382- CE CD 03      0670          DEC NUMB
0385- F0 08         0680          BEQ RESET
                    0690 ; < 12 REPEATS
0387- 4C B4 03      0700          JMP FINISH
                    0710 ; SECOND TIME
038A- A9 FF         0720 SKIP     LDA #$FF
038C- 8D CC 03      0730          STA FIRST
                    0740 ; PICKUP > 12 TIMES
038F- A9 0C         0750 RESET    LDA #12
0391- 8D CD 03      0760          STA NUMB
                    0770 ; ACC/16 AND ACC
0394- AD CB 03      0780          LDA PREV
0397- 4A            0790          LSR A
0398- 4A            0800          LSR A
0399- 4A            0810          LSR A
```

```
039A- 4A        0820            LSR A
039B- 2D CB 03  0830            AND PREV
                0840 ; CONVERT TO ASCII
039E- AA        0850            TAX
039F- BD BB 03  0860            LDA TABL,X
                0870 ; ZERO? RETURN
03A2- F0 10     0880            BEQ FINISH
                0890 ; OTHERWISE STORE IN KEYBOARD BUFFER
03A4- AE 0D 02  0900            LDX $20D
03A7- 9D 0F 02  0910            STA $20F,X
03AA- E8        0920            INX
                0930 ; WATCH FOR BUFFER OVERFLOW
03AB- E0 0A     0940            CPX #10
03AD- D0 02     0950            BNE COUNTER
03AF- A2 00     0960            LDX #$0
03B1- 8E 0D 02  0970 COUNTER    STX $20D
                0980 ; BACK TO NORMAL CODE
03B4- 20 61 03  0990 FINISH     JSR STAX
03B7- EA        1000            NOP
03B8- 4C 7E E6  1010            JMP $E67E
                1020 ;
03BB- 00 00 00  1030 TABL       .BY 0 0 0 '5' 0 '798'
03BE- 35 00 37
03C1- 39 38
03C3- 00 31 33  1040 TABL1      .BY 0 '132' 0 '46' 0
03C6- 32 00 34
03C9- 36 00
03CB- 00        1050 PREV       .BY 0
03CC- 00        1060 FIRST      .BY 0
03CD- 00        1070 NUMB       .BY 0
                1080            .EN
```

CHAPTER TWO: Programmer's Corner



How To Program In BASIC With The Subroutine Power of FORTRAN

Elizabeth Deal

Fortran and PL/I programmers may take subroutines out of the library and append them easily to any main program. Those of us programming in BASIC have to be very careful when we append previously written subroutines, since the variable names have to be identical, rather than passed through a COMMON storage area or a parameter list (as in Fortran). This article describes a way to simulate a COMMON storage area when programming in BASIC. It is written for a upgrade PET, but can be modified for an old PET or adapted for other microcomputers, such as Apple or Ohio Scientific. It requires some extra code which occupies about 2500 bytes, but is is fast and can handle all PET-supported n-dimensional arrays of strings and arithmetic values.

Subroutines in BASIC are internal procedures in that variables active in the main program interact with those in the subroutine. In order to pass parameters to and from a subroutine that has previously been written for another application, we must either rewrite the subroutine or assign all names from the calling program to names used by the subroutine. For single variables it is a nuisance, but easy to do. It is more complicated for arrays. Several options are available: (1) rewrite the subroutine or the main program, (2) move one element of the array at a time, (3) move the entire array at one time if the subroutine requires presence of the whole array. We'll avoid these unwieldly options.

To illustrate the problem, imagine that you have written a subroutine to find minimum and maximum values of a 1000 element array S, but the main program uses array M. Perhaps your subroutine uses S so many times that you dread the idea of rewriting it, or the main program calls the subroutine for many different variables. To use the appended subroutine, the main program will assign a dimension of 1000 elements to arrays M and S. It will then move all elements of M array into S array and then

use the subroutine. The trouble with this procedure is that two thousand elements are used where only one thousand are needed and it takes eight seconds to move the array one way while BASIC interprets the instructions in machine code one thousand times. Were we to pass one value at a time, the memory area would be saved, but the program still would have to spend eight seconds moving the array element into a single variable.

My solution to this memory and time wasting lies in moving *only the name of the array* instead of the array elements. The array stays put. It is never moved, but its name is changed back and forth in about half a second.

The procedure to use for the entire project is to write the main program, append whatever subroutines one needs (via Toolkit or any merge or append program), initialize single variables, append the routine presented here (from line 2520 to 3680), and insert the linking information between two dotted lines in the main program.

More than one variable can share a name with the name used in the subroutine (see array MK% and M%). The number of such variables used in one program is limited only by what can fit in 255 bytes of string QS$. String QS$, together with the DIM statement, is the key to the program. It is a directory of array names in the order in which they appear in the DIM statement. For instance:

Position in the QS$ string: 1 2 3 4 5 6 7
QS$ string contains: M%, MK%, M, MQ$, SI%, S, SQ$
DIM statement contains: M%, MK%, M, MQ$.

In this example, names that begin with "S" refer to arrays used in the subroutine. The asterisk is used to emphasize that they *must not* appear in the DIM statement. They should also not be dimensioned in your subroutine. Actually, it is delightful to get a REDIMENSIONED ARRAY ERROR here, since it is the proof that the name change works.

PET provides us with all the information we need to be able to change names. Pointer to the start of arrays is stored in locations 44 and 45 (125 and 127 in original PET). Pointer to the end of arrays is in location 46 and 47 (128 and 129 for original PET). Array names are stored in the first two bytes preceding each array, and the memory size required by that array is in bytes three and four. In the initializing routine, several Q-arrays are declared. The end of those arrays becomes the pointer to the beginning of the arrays that we are concerned with — those in the DIM statement between two dotted lines. The program (lines 2720 to 2840) reads a name of an array and stores each character, separately, and the address, in a

table. The pointer is then moved to the next array by the number of bytes the first array used. When the names and address of arrays in common have been processed, the program modifies the names in the QS$ string (lines 2960 to 3100) so that each byte will correspond to the internal format of the PET name storage. The two lists are then compared (lines 3100 to 3200) byte by byte and, if they match, the program continues. If there is any discrepancy, it is indicated in the error message and the job is abandoned. (If, during the execution, the arrays are moved, the job will also be abandoned and the program will count the byte shift. However, if it happens in a spot undetectable by my routines, the standard PET error messages will result.)

In order to tell the find + set routines which names to change, we use position numbers in the GS$ string. Thus, to replace name M with S we will set variable QM=3 which corresponds to the third name in the QS$ list, and variable QS=6 which corresponds to the sixth name in the QS$ list The find + set routine will put a name referred to by the QS variable into the name referred to by the QM variable, or S over M. Now the work subroutine uses S, and, on return, the original name is reset so that the calling program can continue whatever it is to do with array M.

The output of the demonstration program consists of showing what happens to the variables at any given time. It occupies about a screenful. Two trivial subroutines are used. #1 adds one to integers, finds MIN and MAX of a floating point array and adds two characters to the elements of string array. #2 creates an array in the subroutine and passes those values to the main program, showing that the process works in both directions and that it does not matter where the array is created.

To make it easier to understand the listing, I have named all variables that are related to or used in the initialize, find + set, and reset routines with a letter "Q" in the first position. Non-Q variables are general variables used in the main program and mathematical routines.

There are several important restrictions that must be observed. Most "BAD SUBSCRIPT ERROR," or "ILLEGAL QUANTITY ERROR" messages occurring during the find + set routine and most of the erroneous data subsequent to return from reset routine will be due to failure to comply with these rules:

1. All single variables and functions (from the main program and subroutines) *must* be assigned a value prior to the DIM statement in the main program and prior to using undimensioned, small arrays.

This is a good rule to follow anyway. Commodore advises that arrays are actually moved seven bytes each time a new single variable is defined in a program. Our program depends on arrays staying in one place throughout the entire program and subroutines. I have not yet found a simple way to have a really independent COMMON area.

2. Provide a string (QS$) of one or two character array names that will be used in common. The array name used in a calling program *must* precede the corresponding names in the subroutine. In the demonstration program M% and MK% precede the listing of SI%, but M% and MK% need not be adjacent to SI% or each other in the QS$ string.

3. Arrays from the main program that will be subject to name change *must* be dimensioned first. They *must* be contiguous. They must be in the same order in which they were listed in the QS$ string. The type of arrays that are common between the calling program and the subroutine *must* agree (i.e., integer with integer), the number of dimensions *must* agree, and their size must agree only to the extent required by the logic of the program. (If array M was dimensioned to twenty elements, but the subroutine tried to use an array S of fifty, you'd end up with a subscript error or garbage. If the array was dimensioned to fifty, and the work subroutine used twenty, there is no problem.)

4. Just before going into the find + set subroutine, tell the program (by use of QM and QS variables) which n ames will be changed. QM must be smaller than QS.

The initialize, find + set and reset routines take up about 2500 bytes at execution time. That is equivalent to two 250 element arrays of floating point numbers. From the memory point of view, it makes sense to use these routines only when the array size exceeds that amount. From the time point of view, it makes sense to use them when the array size exceeds fifty elements. It takes half a second to change and reset the name, but it takes two seconds to move the array one way and two seconds to move it back. From the editing point of view, it is, for me, easier to use this procedure than to fool around with names. The hassle of defining single variables is made easier by PAICS Programmer's Toolkit DUMP command.

Note on terms used in this article:

Fortran programmers should note that this system simulates COMMON in its ability to pass the arrays in BASIC, bu;t that it is based on the concept of EQUIVALENCE, and can be used as such.

PL/I programmer will see that it is similar to the DEFINED (and not BASED) attribute of variables, and that the allocation is still STATIC.

```
1000 REM==============================
1020 REM SIMULATION OF COMMON IN BASIC
1040 REM                BY
1060 REM         ELIZABETH DEAL
1080 REM 337 W.FIRST AVE, MALVERN, PA
1100 REM     19355; (215)647-4876
1120 REM       APRIL 9, 1980
1140 REM REF:
1160 REM 1) GENE BEALS OF AB COMPUTERS
1180 REM 2) PET USER NOTES 3) COMPUTE
1200 REM 4) LEN LINDSAY :OLD PET/NEW
1220 REM     PET IN MICROCOMPUTING
1240 REM 5) COMMODORE MANUAL
1260 REM==============================
1280 REM
1300 REM---DEFINE NON-Q SINGLE-V,FN---
1320 IN=20:FL=15:LI$="":FORI=1TO39:
      ¬LI$=LI$+"-":NEXTI:JI=0:JF=0:Q=0:
      ¬KX=0
1340 MN=0:MX=0:I=0:J=0:K=0:S1=2:S2=2:
      ¬S3=2:Z=64:L=0:M=0:N=0:TX=0
1360 REM
1380 REM.............................
1400 :
1420 REM LINK MAIN WITH Q SUBROUTINES
1440 REM DEFINE QS$,INITIALIZE IN THE
1460 REM SUB, DIM ARRAYS IN COMMON,
1480 REM BACK TO INIT., THEN MAIN....
1500 :
1520 QS$="M%,MK%,M,MQ$,SI%,S,SQ$"
1540 GOSUB2520
1560 DIM M%(IN),MK%(IN),M(FL)
1580 DIM MQ$(S1,S2,S3)
1600 GOSUB2720
1620 :
1640 REM.............................
1660 REM
1680 REM   BACK TO MAIN PROGRAM
1700 REM
1720 REM---DIM ALL OTHER ARRAYS-------
1740 REM          (NONE HERE)
1760 REM---ASSIGN VALUES--------------
1780 FORJ=1TOIN:MK%(J)=J:NEXT
1800 FORJ=1TOFL:M(J)=-J/100:NEXT
1820 FORI=1TOS1:FORJ=1TOS2:FORK=1TOS3:
```

```
      ¬MQ$(I,J,K)=CHR$(I+Z)+CHR$(J+Z)+CHR
      ¬$(K+Z)
1840 NEXTK,J,I
1860 REM---ASSIGN QS,QM FOR NAME
1880 REM    CHANGE; POKE NEW NAME; USE
1900 REM    IN SUBROUTINES; RESET NAME,
1920 REM    CONTINUE.
1940 REM
1960 PRINTLI$: REM DEMO FEW ARRAYS----
1980 QS=7:QM=4:GOSUB3300:QS=6:QM=3:
      ¬GOSUB3300:QS=5:QM=2:GOSUB3300:
      ¬REM FIND+SET
2000 JI=IN:JF=FL:I=S1:J=S2:K=S3:
      ¬GOSUB2260:PRINT"SUB#1, ";
2020 GOSUB3460:REM RESET ALL NAMES----
2040 PRINT:PRINT"(MK%)";:FORJ=1TOIN:
      ¬PRINTMK%(J);:NEXT:PRINT
2060 PRINT:PRINT"(M  )";:FORJ=1TOFL:
      ¬PRINTM(J);:NEXT:PRINT
2080 PRINTTAB(10)"MIN="MN", MAX="MX
2100 PRINT:PRINT"(MQ$)";:FORI=1TOS1:
      ¬FORJ=1TOS2:FORK=1TOS3:PRINTMQ$(I,
      ¬J,K);
2120 NEXTK,J,I:PRINT:PRINTLI$
2140 REM            DEMO ONE ARRAY-----
2160 QS=5:QM=1:GOSUB3300:KX=IN:GOSUB2380
      ¬:PRINT"SUB#2, ";:GOSUB3460
2180 PRINT:PRINT"(M%)";:FORJ=1TOIN:
      ¬PRINTM%(J);:NEXT:PRINT:PRINTLI$
2200 END:  REM END MAIN
2220 REM
2240 REM===WORK SUB#1================
2260 ::FORQ=1TOJI:SI%(Q)=SI%(Q)+1:
      ¬     NEXTQ
2280 FORL=1TOI:FORM=1TOJ:FORN=1TOK:
      ¬SQ$(L,M,N)="*"+SQ$(L,M,N)+"/":
      ¬NEXTN,M,L
2300 MX=-1.7E38:MN=1.7E38:FORJ=1TOJF:
      ¬IFS(J)<=MNTHENMN=S(J)
2320 IFS(J)>=MXTHENMX=S(J)
2340 NEXTJ:RETURN
2360 REM===WORK SUB#2================
2380 ::FORK=1TOKX:SI%(K)=K:NEXT:RETURN
2400 REM
2420 REM==*==*==*==*==*==*==*==*==*==*
2440 REM
2460 REM===SUB: INITIALIZE============
2480 REM
2500 REM---DCL Q'S: SINGLE,FN,ARR-----
2520 ::PRINT"INIT1":DEFFNQA(QP)=PEEK(QP)
```

```
      ¬+256*PEEK(QP+1)
2540 QB=0:QC=0:QD=0:QE=0:QF=0:QH=0:QI=0:
      ¬QJ=0:QK=0:QL=0:QM=0:QN=0
2560 QP=0:QQ=0:QR=0:QS=0:QT=0:QU=0:QW=0:
      ¬QX=0:QY=0:QZ=0:Q=0
2580 QK$="":QM$="":QL=LEN(QS$)/2
2600 IFPEEK(50003)=1THENQ5=44:Q6=46:
      ¬Q7=54:Q8=152:GOTO2640
2620 Q5=126:Q6=128:Q7=136:Q8=516
2640 ::DIMQG(QL),Q1(QL),Q2(QL),QD$(QL),
      ¬Q3(QL),Q4(QL),QN$(QL):FORQ=1TOQL
2660 QD$(QL)="":QN$(QL)="":NEXT
2680 QZ=FNQA(Q5):QI=FNQA(Q6):PRINT"MAIN"
      ¬:RETURN
2700 REM---ARRAYS FROM DIM------------
2720 ::PRINT"INIT2":QD=FNQA(Q7):
      ¬GOSUB3640:QH=QI:QE=FNQA(Q6)
2740 ::QK=QK+1:QG(QK)=QH:QC=QG(QK):
      ¬Q1(QK)=PEEK(QC):Q2(QK)=PEEK(QC+1)
2760 QX=Q1(QK):QY=Q2(QK)
2780 QT=0:QT=QT+37*(-(QX>=128ANDQY>=128)
      ¬)+36*(-(QX<128ANDQY>=128))
2800 QX=QX-128*(-(QX>128)):QY=QY+128*(-(
      ¬QY=0))-128*(-(QY>128))
2820 QD$(QK)=CHR$(QX)+CHR$(QY)+CHR$(QT):
      ¬REM DO NOT COMPARE QD$ TO QN$ !!
2840 QN=FNQA(QC+2):QH=QH+QN:IFQH<QEGOTO2
      ¬740
2860 REM---ARRAYS FROM QS$------------
2880 QU=1:FORQ=1TOLEN(QS$):QK$=MID$(QS$,
      ¬Q,1):IFQK$=","THENQU=QU+1:NEXTQ
2900 QN$(QU)=QN$(QU)+QK$:NEXTQ
2920 IFQU<2THENPRINT"***CORRECT QS$":END
2940 REM---FIX NAMES IN QS$-----------
2960 FORQ=1TOQU:QM$=QN$(Q):QF=0:QJ=0:
      ¬QT=ASC(RIGHT$(QM$,1))
2980 QT=QT*(-(QT=36ORQT=37)):QF=128*(-(Q
      ¬T=37)):QJ=128*(-(QT=37ORQT=36))
3000 IFLEN(QM$)>2GOTO3060
3020 IFQJ=0ANDQT=0THENQM$=QM$+CHR$(0):
      ¬GOTO3060
3040 QM$=LEFT$(QM$,1)+CHR$(128)+CHR$(QT)
3060 ::QX=QF+ASC(LEFT$(QM$,1)):QY=QJ+ASC
      ¬(MID$(QM$,2,1)):QY=QY-QJ*(-(QY>255
      ¬))
3080 Q3(Q)=QX:Q4(Q)=QY:REM PRINTQ;TAB(4)
      ¬QM$;TAB(10)QX;QY
3100 NEXTQ:QQ=0:FORQ=1TOQK:IFQ1(Q)=Q3(Q)
      ¬ANDQ2(Q)=Q4(Q)GOTO3140
3120 QQ=QQ+1:QN$(Q)=QN$(Q)+"  <-- ??"
```

```
3140 ::NEXTQ:IFQQ=0GOTO3200
3160 PRINT:PRINT"***NO MATCH !":PRINT:
      ¬PRINT"QS/QM  QS$"TAB(20)"DIM":
      ¬PRINT
3180 FORQ=1TOQU:PRINTTAB(1)QTAB(7)QN$(Q)
      ¬TAB(20)QD$(Q):NEXTQ:END
3200 PRINT"MAIN.":PRINT"PRESS 'SHIFT' ¬
      ¬TO CONT MAIN PRGM":WAITQ8,1:
      ¬PRINT" OK"
3220 RETURN
3240 REM
3260 REM===SUB: FIND+SET NAME=========
3280 REM
3300 ::QD=FNQA(Q7):GOSUB3640
3320 IFQS<=QMTHENPRINT"***BAD POSITION ¬
      ¬NUMBERS":PRINT"QS="QS", QM="QM:END
3340 QR=QG(QM):PRINT"SET "QN$(QS)TAB(8)"
      ¬OVER "QN$(QM);:PRINTTAB(18)"AT"QR
3360 QB=QB+1:QG(QM)=-QG(QM):POKEQR,
      ¬Q3(QS):POKEQR+1,Q4(QS)
3380 RETURN
3400 REM
3420 REM===SUB: RESET NAME===========
3440 REM
3460 ::IFQB<1THENPRINT"***IMPROPER CALL ¬
      ¬TO RESET":END
3480 QD=FNQA(Q7):GOSUB3640:PRINT"RESET ¬
      ¬NAME AT";
3500 FORQ=1TOQK:IFQG(Q)>0GOTO3540
3520 QG(Q)=-QG(Q):QR=QG(Q):POKEQR,Q1(Q):
      ¬POKEQR+1,Q2(Q):PRINTQR;
3540 ::NEXTQ:PRINT:RETURN
3560 REM
3580 REM---CHECK COMMON ARRAYS POS----
3600 REM   CALLED BY INIT,SET,RESET
3620 REM
3640 ::QW=FNQA(Q5):IFQW=QZTHENRETURN
3660 PRINT: PRINT"***ARRAYS MOVED"QW-QZ"
      ¬BYTES"
3680 PRINT"   PRIOR TO LINE"QD:END
3700 REM=============================
```

Sorting Sorts: A Programming Notebook Part One

Belinda and Rick Hulon

An important aspect of many business applications involving microcomputers is the selection and efficient use of an appropriate sorting routine. While sorting routines are readily available in the literature and/or easily programmable, some thought should be given to the type of sort used. This two-part article will deal with six of the better known sorting algorithms: Selection Sort, Bubble Sort, Shell Sort, Quick Sort, Merge Sort, and Heap Sort. These sorts range from very simple to quite complex, from extremely slow to exceedingly fast. This first article will concern itself with the simpler, slow to intermediate sorts.

In this article we evaluate Selection Sort, Bubble Sort and Shell Sort. Selection Sort is a very simple, straightforward routine (see Figure 1). Its method is to iteratively pass through the list of items to be sorted. On the initial pass, the first item is compared with each successive item, exchanging it with any element that is "less than" the first item. The "new" first element is then compared to each item after the point of exchange. This process continues until one entire pass is completed. The procedure is then repeated for the second item in the list, then the third, etc., until the last item is reached. Thus, Selection Sort essentially "selects out" the smallest item on the first pass, the next smallest on the second, and so on. This sort, then, always goes through a set number of passes regardless of the state of the list. An already sorted list would still go through the entire routine as though it were not sorted.

Bubble Sort accomplishes its task not by comparing one item to all the others as in Selection Sort, but rather by comparing adjacent elements in the list, switching whenever necessary. In addition, it sets a "flag" to indicate when no exchanges have been made in a given pass, thus signalling the end of the sort. Bubble Sort, therefore, takes only as many passes as it needs. An already sorted list would use one pass to determine that no exchanges were made. A listing of Bubble

Sort can be found in Figure 2.

The third sort to be considered, Shell Sort, is somewhat more complicated. Shell Sort is essentially an extension of Bubble Sort. Initially a "gap" size is determined to be the largest integer less than or equal to half of the list size (e.g., if the list contained 11 items, the initial gap size would be 5). This gap size supplies the essential difference between Shell Sort and Bubble Sort, for, instead of only comparing adjacent items, Shell Sort compares items separated by the gap size, exchanging them when necessary. Once it determines that no exchanges were made on the last pass with a particular gap size, the size of the gap is cut in half and the process continues. As one can easily see, this results in a Bubble Sort when the gap size becomes 1, but since the list is already partially sorted, it does not require as much time for larger lists as a regular Bubble Sort would. Shell Sort, like Selection Sort, does not provide for an "easy out." However, it does not go through the routine a set number of times: a pre-sorted list will require only enough passes to obtain a gap size of 1, since there will never be any exchanges. (See Figure 3)

```
10 FOR I=1 TO N-1
20 FOR J=I+1 TO N
30 IF V(I) <= V(J) THEN 70
40 S=V(I)
50 V(I)=V(J)
60 V(J)=S
70 NEXT J
80 NEXT I
90 END
```

Figure 1

```
10 F=1
20 IF F=0 THEN 120
30 F=0
40 FOR I=1 TO N-1
50 IF V(I)<=V(I+1) THEN 100
60 S=V(I)
70 V(I)=V(I+1)
80 V(I+1)=S
90 F=1
100 NEXT I
110 GOTO 20
120 END
```

Figure 2

```
10 GP=N
20 IF GP<=1 THEN 160
```

```
30 GP=INT(GP/2)
40 MI=N-GP
50 F=0
60 FOR I=1 TO MI
70 GI=GP+I
80 IF V(I)<=V(GI) THEN 130
90 S=V(I)
100 V(I)=V(GI)
110 V(GI)=S
120 F=1
130 NEXT I
140 IF F=1 THEN 50
150 GOTO 20
160 END
```

Figure 3

```
WHERE:
V =Array containing data to be
   sorted
S =Holding variable used for
   exchanging items
N =Number of items to be
   sorted
F =Flag indicating occurrence
   of an exchange
GP =Gap size
MI =Number of times the loop
    must be iterated on one
    pass; depends on gap size
GI =Index for comparison;
    depends on gap size
```

Several factors are involved in determining the efficiency of a sorting algorithm. The time involved, or the speed of the sort, is usually one of our major concerns, especially with micros. Two important factors contributing to the speed and efficiency of a sort are the number of comparisons made and the number of actual exchanges involved in sorting a list. In this case we counted any comparison made (including the checking of flags), not just data comparisons. Although we are not directly concerned with CPU time in terms of actual cost, it seems obvious that the few comparisons and exchanges made in the same (or less) amount of time, the more efficient the sort will be. These three factors then, time, number of comparisons, and number of exchanges comprise our criterion for comparison. The actual method used was to generate 30 different lists of random numbers, having each algorithm sort each list. The 30 values for each of the criterion for the different list sizes were averaged to produce the values given in Table 1. This

procedure was followed for lists of size 10, 25, 50 and 100. All data was obtained from a Commodore CBM with 32K of internal RAM.

TABLE 1

	List Size	Number of Comparisons	Number of Exchanges	Time
BUBBLE SORT				
	10	75	21	1.8s
	25	508	153	12.4s
	50	2098	592	50.7s
	100	8811	2450	3.6m
SELECTION SORT				
		45	21	1.1s
		300	144	7.3s
		1225	505	28.0s
		4950	1815	1.8m
SHELL SORT				
		72	11	1.5s
		339	63	7.4s
		967	155	20.9s
		2669	399	57.2s

WHERE:
s = seconds
m = minutes

It should be noted that upon beginning this article there were some basic expectations. Having already run a similar project on a large computer, we expected similar results from the CBM. The initial project showed, true to the numerous textbooks available, that while Selection Sort and Bubble Sort were good for small lists (even superior to more sophisticated sorts), Shell Sort would be better for larger lists. Also, Selection Sort should be faster than Bubble Sort, due to the nature of the algorithms (we omit the mathematical determination of this situation). In this experiment we did duplicate our first results fairly well, as can be seen from Table 1. However, the amount of time involved seemed flabbergasting. Of course, we could not have expected a micro to compare in speed to a mainframe, but the differences were disturbing. For example, the time involved in the sorting of a list of 500 items by one of these sorts ran into hours, somewhat troublesome for business applications. Having mulled over this for a while we came to a tentative conclusion which seemed to explain this occurrence. Our original sorting routines were written in PL/1, a batch language for use on an IBM 360/370. In this situation the source code went through a compiler which translated it into machine code for execution. On our CBM, however, the routines were written in interpreted BASIC. One important difference

between an interpreter and a compiler is that, with a compiler, the source is "compiled" only once. The machine code is produced and the higher level language is no longer a concern. With an interpreter, on the other hand, each line of code is interpreted *every* time it is encountered. This, then, should account for much of the excessive time observed. As a test, we wrote Selection Sort (chosen for its simplicity) in machine code. This eliminated the interpretation stage. This gives us a better idea of just how much time is actually involved in sorting the lists. The elimination of an interpreter changed the time involved drastically. While the BASIC routine required 1.1 seconds to sort a list of only 10 items, the machine code version took

TABLE 2

List Size	Time for BASIC routine	Time for mach. code routine
10	1.1 sec	0.00 sec
25	7.3 sec	0.02 sec
50	28.0 sec	0.05 sec
100	1.8 min	0.17 sec

so little time as to record a duration of 0 seconds! Since the built-in timer of the CBM records time in "jiffies" or 1/60 of a second, it actually took less than 1/60 of a second to sort the list. The results are even more impressive for a list of 100 items. While BASIC required 109.2 seconds (just under 2 minutes) the machine code version required only .2 seconds. In other words, the BASIC algorithm took 546 times longer than did the machine code routine. Much of this extra time, then, seems to be a result of the BASIC interpreter. Thus, what might seem to be a very efficient sort could actually prove to be worse than a less efficient sort, depending on the amount of code involved and the number of items to be sorted. In the design of business software, as much attention should be paid to the language (and therefore type of compiler or interpreter) as to the type of sort involved. If you are willing to work with machine code then more efficient sorts should be considered. We will include machine code listings in the next article along with the evaluations of Quick Sort, Merge Sort, and Heap Sort.

Part Two

As mentioned in the first article of this two-part series, the selection of an appropriate sorting algorithm is crucial for many business applications involving microcomputers. While the first article concerned itself with the slow to intermediate sorts (Selection, Bubble and Shell), this article deals with the faster, more sophisticated (and therefore less intuitive) algorithms. At the outset of the writing of this article three "fast" sorts were under consideration: Quick Sort, Heap Sort, and Merge Sort. Initially Merge Sort was thought to be an appropriate sort since it is not only fairly fast, but also is the one chosen as the "built-in" sorting algorithm for many mainframes. Upon closer examination it was determined that Merge Sort was not a viable algorithm for a micro (at least not the version to which we had access). While the actual programming could be done, the routine would require an immense amount of data storage and numerous array swaps. Since this algorithm was of dubious value for business applications on a micro, we decided to delete it from the article. This article will instead concentrate on the comparison of Heap Sort and Quick Sort as well as relating them to the previously examined sorts. It will also discuss the advantages of a machine code algorithm over its BASIC counterpart and present a hopefully usable machine code version of Heat Sort for your implementation.

Quick Sort is a fairly fast sorting algorithm which achieves its goal by subdividing the original list of data items. This is done by initially placing the first item in the list in its proper place relative to the other items in the list, i.e., all of the items to its "right" are larger. This process continues with the two newly created lists until the entire array is sorted. Quick Sort, though complicated, is a very efficient sort.

Heap Sort is an even more complicated algorithm which involves the use of a "binary tree". The sort is achieved by "traversing" the tree. The larger items are worked up a "branch," one by one, until they reach the top. Each is then placed in its appropriate place at the bottom of the heap and a new value "climbs" the tree. While this algorithm is even less intuitive than Quick Sort, it, too, is a very speedy algorithm.

It is important to note that, while a thorough understanding of these algorithms would be helpful in terms of making modifications, etc., it is not crucial to the implementation of the listings provided as useful tools. The above descriptions are obviously not intended

to provide you with a complete understanding of how these routines work. Rather, they are simply meant to give you a general idea of their functioning. There are numerous books on the subject of sorting which would provide you with a better understanding of these algorithms. Since such a discussion would be lengthy, is not the purpose of this article and is not necessary to use the presented routines, we will not engage in a further description of the algorithms.

In the last article, time, number of comparisons, and number of exchanges were used as the basis of comparing the sorting algorithms. Due to the nature of the routines currently being scrutinized, it becomes difficult to define comparisons and exchanges. Certainly the end result would not be something one could compare to the more straightforward comparison and exchange counts of the slower algorithms. For that reason, this article shall concentrate on the less questionable concept of time. Indeed, for most users this will be the most important factor anyway.

For consistency, the data was gathered in the same manner as before: thirty lists of random numbers were generated, sorted and timed. The times were then averaged to give the data reported in Table 1. This process was repeated for lists of size 10, 24, 50, 100, 500.

Upon examining the data in Table 1, it is easy to see that Quick Sort was consistently faster than Heap Sort. Our prediction was that Heat Sort would be the faster sort on larger lists, but it did not hold true. However, one must remember that, unlike the sorts in the last article, there are numerous versions of Quick Sort and Heap Sort and Heap Sort available. Consequently, the results could vary depending on the rendition used. For our particular versions, Quick Sort became increasingly faster than Heap Sort as the list size increased. Looking back at the data collected for the three slower sorts, one can readily verify that Quick Sort and Heap Sort are much more efficient. For lists of ten, the "fastest" time recorded for the "slow" sorts was 1.1 seconds (Selection Sort). From Table 1 we can see that Quick Sort required .99 seconds and Heap Sort 1.31 seconds to sort ten items. This is not surprising since Heap Sort is known to be more efficient on larger lists. Indeed, as the list size increases, both Heap Sort and Quick Sort out-perform the others. For lists of size 100, Quick Sort required 16.2 seconds and Heap Sort 25.17 seconds; the least time recorded for the slower sorts was 57.2 seconds by Shell Sort. In fact, Selection Sort required 1.8

minutes and Bubble Sort 3.6 minutes to sort 100 items. One might also note that while it took Quick Sort an average of 105.12 seconds to sort 500 items, it was not even feasible to sort 500 items by the slower sorts due to the immense amount of time involved.

It was shown in the last article that, due to the nature of an interpreter (vs. a compiler), BASIC algorithms are much slower than their counterparts in a compiled language (ex. PL/1). Since an interpreter must translate each BASIC statement into machine code every time it is encountered, an algorithm written in machine code should proceed much faster. Indeed, the data presented in the previous article showed that a machine code version of Selection Sort (a "slow") sorted 100 items in 0.17 seconds! Even Quick Sort required as much as 16.2 seconds to sort a list of this size. In fact, Quick Sort required .99 seconds to sort only ten items. It took Quick Sort almost six times longer to sort ten items than the machine code version of Selection Sort to sort 100 items. It should be obvious then that a machine code version of even a slow sort would be preferable to a BASIC version of a fast sort. While the BASIC algorithms presented were capable of sorting random numbers of more than one byte, only random numbers between 1 and 100 were used so that each number was only one byte in length. The machine code sort was very specialized in that it would only sort 255 or fewer single byte data items. For this reason no listing is presented. Included, however, is a machine code version of Heap Sort which will sort more than one byte of data. The data provided in Table 2 was obtained by sorting character strings of 10 or fewer characters (bytes). The numbers cannot be directly compared to the other sorts since the algorithm does sort larger items of data, but, even so, one can easily see that it is far superior to the BASIC algorithms.

The listing provided of the machine code version of Heap Sort is actually a BASIC program containing the sorting routine in DATA statements. Lines 110-190 of the program poke the algorithm into memory beginning at hex location $1600. Please note that this routine is not relocatable, therefore it will not be possible to alter its starting position. The BASIC program as listed is very similar to the one we used to gather the data that appears in Table 2 in that it generates N random character strings with length varying from one to ten bytes and sorts those items. We suggest that if you intend to use this sorting algorithm you first copy the given listing verbatim. In that way you will very easily be able to determine if indeed you have copied the entire list of data statements properly by

simply running the program. Be sure, however, to save the program before you attempt to execute it, for if any data statements were copied incorrectly you could easily lose control of the execution and be forced to implement a cold start. Once you are sure the algorithm is functioning correctly, you may proceed to modify the program to suit your particular need.

As an example of how you might use this algorithm, suppose you have 100 character strings stored on tape that you wish to have sorted and printed on the screen. You could accomplish this task by making the following additions and substitutions:

```
90 OPEN 1,1,0,filename
350 INPUT#1,A$
360 L = LEN(A$)
420 X$ = MID$(A$,J,1) :PRINT X$;
440 POKE BD + P + J,ASC(X$)
```

After keying in the above lines, it is simply necessary to run the program and respond to the list size prompt with "100". As another example, suppose the situation were the same as the above, but instead of character strings, the tape file contained monetary amounts in the range $0.00 to about $65,000.00. The following changes should be made.

```
90 OPEN 1,1,0,filename
350 INPUT#1,A:PRINT A
360 L = 3
400 A2 = INT(A)
410 Q = INT(A2/256)
420 R1 = A2-Q*256
430 R2 = INT( (A-A2) *100 + 100 + .5)
440 POKE BD + P + 1,Q
450 POKE BD + P + 2,R1
460 POKE BD + P + 3,R2
500
800 A = 256*PEEK(Y + 1) + PEEK(Y + 2)
820 A = A + (PEEK(Y + 3)/100)
840 PRINT A
860
880
```

In total, on a 32K machine, this algorithm is equipped to handle approximately 5000 bytes of data. It will function on 8K, 16K, and 32K Commodore CBMs and PETs, but of course the amount of data that can be used will depend on the amount of RAM available.

This two-part series of articles, then, has concerned itself with the selection of an appropriate sorting algorithm. Listings of slow, fast, simple and complex sorts have been provided and at least

partially explained. It has been pointed out that such criteria as the time involved and the amount of knowledge and actual programming required must always be of concern to the user when choosing a sorting routine. Depending on the type of application needed and the expertise of the user, any of the BASIC algorithms or the machine code sort that we have presented could be implemented. Microcomputers can be of vast benefit to the small business, but only if used properly and wisely. The selection of an appropriate sorting algorithm is an important part of using your micro wisely.

List Size	Avg. time for Quick Sort	Avg. time for Heap Sort
10	.99	1.31
25	3.09	4.37
50	7.11	10.69
100	16.21	25.17
500	105.12	169.91

Time in seconds
TABLE 1

List Size	Time (secs) for mach. code Heap Sort
10	.09
25	.14
50	.23
100	.45
500	2.67

TABLE 2

```
HEAPSORT

100 L=INT(N/2)+1
120 K=N
140 IF L=1 THEN 220
160 L=L-1
180 S=V(L)
200 GOTO 300
220 S=V(K)
240 V(K)=V(1)
260 K=K-1
280 IF K<1 THEN V(I)=S:GOTO 440
300 J=L
320 I=J
340 J=J+J
360 IF J>K THEN V(I)=S:GOTO 140
```

```
380 IF J<K THEN IF V(J)<V(J+1) THEN J=J+1
400 IF S>=V(J) THEN V(I)=S:GOTO 140
420 V(I)=V(J):GOTO 320
440 END
```

QUICK SORT

```
100 TP=1:LOWER(1)=1:UPPER(1)=N
120 IF TP<=0 THEN 480
140 LB=LOWER(TP):UB=UPPER(TP):TP=TP-1
160 IF UB<=LB THEN 120
180 I=LB:J=UB:TEMP=V(I)
200 IF J<1 THEN 260
220 IF TEMP>=V(J) THEN 260
240 J=J-1:GOTO 200
260 IF J<=I THEN V(I)=TEMP:GOTO 400
280 V(I)=V(J):I=I+1
300 IF I>N THEN 360
320 IF V(I)>=TEMP THEN 360
340 I=I+1:GOTO 300
360 IF J>I THEN V(J)=V(I):J=J-1:GOTO 220
380 V(J)=TEMP:I=J
400 TP=TP+1
420 IF I-LB<UB-I THEN LOWER(TP)=I+1:
    UPPER(TP)=UB:UB=I-1:GOTO 160
440 LOWER(TP)=LB:UPPER(TP)=I-1:LB=I+1
460 GOTO 160
480 END
```

HEAP SORT IN MACHINE CODE

```
100 PRINT"           ONE";
105 PRINT" MOMENT, PLEASE.
110 BS=5632:BA=6402
120 FOR I=BS TO BS+1000
140 READ A
160 IF A=500 THEN 240
180 POKE I,A
190 NEXT I
240 P=0:C=0:PRINT"
260 INPUT"HOW MANY ITEMS";N
265 PRINT"           THE ORIGINAL LIST
270 BD=BA+2*N+10
280 FOR I=1 TO N
290 PRINTI" ";
300 Q=INT((BD+P)/256)
320 R=(BD+P)-Q*256
340 POKE BA+C,R:POKE BA+C+1,Q
360 L=INT(10*RND(1))+1
380 POKE BD+P,L
400 FOR J=1 TO L
```

```
420 X=INT(26*RND(1))+65:PRINTCHR$(X);
440 POKE BD+P+J,X
460 NEXT J
480 P=P+L+1:C=C+2
500 PRINT
520 NEXT I
540 T1=TI
560 Q=INT(N/256)
580 R=N-Q*256
600 POKE 6168,R:POKE 6169,Q
620 L=INT(N/2)+1
640 Q=INT(L/256)
660 R=L-Q*256
680 POKE 6170,R:POKE 6171,Q
700 SYS 05632
720 T2=(TI-T1)/60
740 PRINT"              THE SORTED LIST
760 FOR I=0 TO N-1
770 PRINTI+1" ";
780 Y=PEEK(BA+2*I)+256*PEEK(BA+2*I+1)
800 L=PEEK(Y)
820 FOR J=1 TO L
840 PRINTCHR$(PEEK(Y+J));
860 NEXT J
880 PRINT
900 NEXT I
920 PRINT""T2" SECS
930 PRINT" HIT ANY KEY TO CONTINUE
940 GET T$:IF T$="" THEN 940
960 GOTO 240

1000 DATA 173,27,24,208,104,173,26,24
1010 DATA 201,1,208,97,173,24,24,24
1020 DATA 42,133,178,173,25,24,42,105
1030 DATA 25,133,179,234,234,160,0,177
1040 DATA 178,141,32,24,200,177,178,141
1050 DATA 33,24,173,3,25,145,178,136
1060 DATA 173,2,25,145,178,56,173,24
1070 DATA 24,233,1,141,24,24,173,25
1080 DATA 24,233,0,141,25,24,201,0
1090 DATA 208,80,173,24,24,208,75,173
1100 DATA 28,24,24,42,133,178,173,29
1110 DATA 24,42,105,25,133,179,234,173
1120 DATA 32,24,160,0,145,178,200,173
1130 DATA 33,24,145,178,96,56,173,26
1140 DATA 24,233,1,141,26,24,173,27
1150 DATA 24,233,0,141,27,24,173,26
1160 DATA 24,24,42,133,178,173,27,24
1170 DATA 42,105,25,133,179,160,0,177
1180 DATA 178,141,32,24,200,177,178,141
1190 DATA 33,24,173,26,24,141,30,24
1200 DATA 173,27,24,141,31,24,173,30
```

```
1210 DATA 24,141,28,24,173,31,24,141
1220 DATA 29,24,24,46,30,24,46,31
1230 DATA 24,173,31,24,205,25,24,240
1240 DATA 5,144,16,76,86,23,173,30
1250 DATA 24,205,24,24,144,5,240,89
1260 DATA 76,86,23,173,30,24,24,42
1270 DATA 133,178,173,31,24,42,105,25
1280 DATA 133,179,234,234,160,0,177,178
1290 DATA 133,180,200,177,178,133,181,24
1300 DATA 165,178,105,2,133,178,165,179
1310 DATA 105,0,133,179,160,1,177,178
1320 DATA 72,136,177,178,72,165,181,72
1330 DATA 165,180,72,32,167,23,104,133
1340 DATA 178,104,197,178,144,19,240,17
1350 DATA 24,173,30,24,105,1,141,30
1360 DATA 24,173,31,24,105,0,141,31
1370 DATA 24,173,30,24,24,42,133,178
1380 DATA 173,31,24,42,105,25,133,179
1390 DATA 160,1,177,178,72,136,177,178
1400 DATA 72,173,33,24,72,173,32,24
1410 DATA 72,32,167,23,104,133,178,104
1420 DATA 197,178,240,2,176,33,173,28
1430 DATA 24,24,42,133,178,173,29,24
1440 DATA 42,105,25,133,179,234,234,160
1450 DATA 0,173,32,24,145,178,200,173
1460 DATA 33,24,145,178,76,0,22,173
1470 DATA 28,24,24,42,133,180,173,29
1480 DATA 24,42,105,25,133,181,234,234
1490 DATA 173,30,24,24,42,133,178,173
1500 DATA 31,24,42,105,25,133,179,234
1510 DATA 234,160,0,177,178,145,180,200
1520 DATA 177,178,145,180,76,166,22,104
1530 DATA 141,34,24,104,141,35,24,104
1540 DATA 133,178,104,133,179,104,133,180
1550 DATA 104,133,181,160,0,177,178,209
1560 DATA 180,240,9,176,14,133,182,162
1570 DATA 1,24,144,13,133,182,162,0
1580 DATA 24,144,6,177,180,133,182,162
1590 DATA 2,160,1,177,178,209,180,208
1600 DATA 38,200,196,182,144,245,240,243
1610 DATA 224,1,240,9,16,16,169,0
1620 DATA 72,72,76,15,24,177,180,72
1630 DATA 169,0,72,76,15,24,169,0
1640 DATA 72,177,178,72,76,15,24,133
1650 DATA 178,177,180,72,165,178,72,173
1660 DATA 35,24,72,173,34,24,72,96
1670 DATA 500
```

Saving Memory In Large Programs:

Mike Richter

If you find your free memory space getting cramped, try some of these suggestions to solve the ?OUT OF MEMORY ERROR.

1. Pack your statements into long lines. Each new line number costs four bytes more than a colon for continuation.

2. On very long lines, use the shorthand (? for PRINT, gO for GOTO) to stay within the 80-character limit.

3. Relace IF X = O THEN A = A + 1 with A = A- (X = O). A logical expression evaluates to -1 if true, O if false. Those values may be used arithmetically.

4. Some IF . . . GOTO structures can be replaced efficiently with ON . . . GOTO. For example:
1000 IF X = 1 GOTO 100
1010 GOTO 200
may be replaced with: 1000 ONXX GOTO 100: GOTO 200

5. Close up the spaces in the BASIC statements; they just waste storage, although they may help readability.

6. Semicolons are rarely needed in single-line printing. For example,
PRINT TAB(5)"X="X
prints the same as PRINT TAB(5);"X=";X

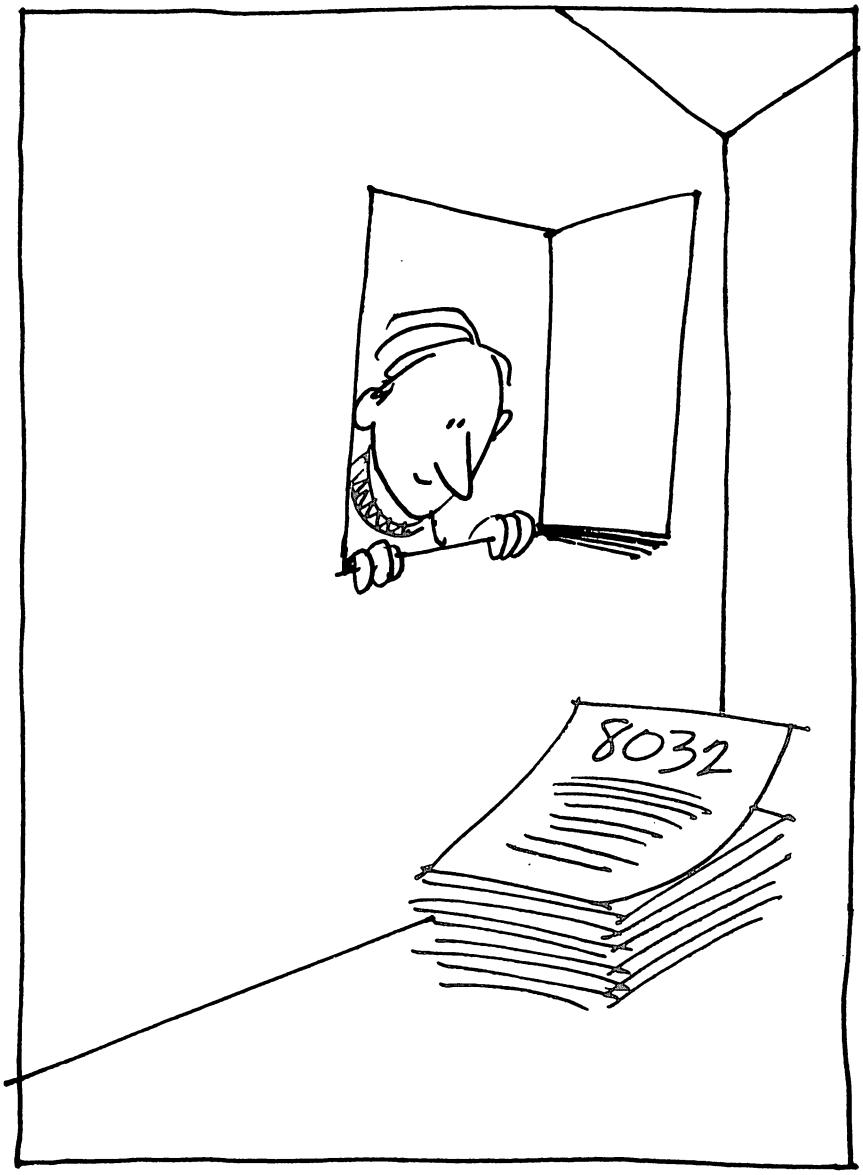
7. Use computed values in TAB and SPC expressions rather than FOR/NEXT.

8. A string of blanks (usually, 39 of them) is useful for erasing all or a part of a line. To erase 20 characters, PRINT LEFT$(BL$,20). Variations and extensions of the idea are numerous. A string of cursor control characters can be used to locate a line in the same way. With CC$ defined as "home, 24xcd", you get to line N by PRINT LEFT$(CC$,N+1).

9. Putting the above material together may save 20-50% of the code. One check you can make on how tightly the program is packed is to figure out what fraction of your lines must end where they do because they either finish with an unavoidable IF or just plain run out of space on the 80-character line.

Don't be fanatical about saving space, but wonders can

sometimes be worked. I took a 24K APPLE program and added features in transferring it to the PET. The result took les than 2.4K of memory! OREGON TRAIL is another example which probably took well over 20K as published, yet runs in the 8K (really, 7K-1) of a PET when properly compressed.



Programmer's Notes For The CBM 8032

Roy Busdiecker

Several good articles describing major features of the CBM 8032, have already appeared (Butterfield Reports: The 8032, by Jim Butterfield, COMPUTE! Issue #5; and New Additions to the Commodore Line, by Robert W. Baker, *Kilobaud Microcomputing*, July 1980). There are quite a few features, however, which were not mentioned in those articles and will be of interest to those who own, or are contemplating purchase, of the new machine.

New Functions from Keyboard

My most recent (and most exciting) discovery is the fact that many of the new screen-editor functions (scroll down, delete line, insert line, etc.) can be activated directly from the keyboard, without the necessity of doing a PRINT CHR$ (XX) as described in the articles. The trick is simply to press the right combination of keys simultaneously. The combinations are shown in Figure 1. In some cases, it doesn't matter which key is pressed first; however, it's generally safer to press the key listed in the left column first.

Abbreviation	Meaning (Key)
DE	Delete
ES	Escape
LA	Left Arrow
LS	Shift Key on Left Side
OR	Off/Reverse
RS	Shift Key on Right Side
SH	Either Shift
TA	Tab
UA	Up Arrow
k	Key on Alpha-Numeric Keyboard
p	Key on Numeric Keypad

Function	Keys
Condensed graphics	LS RS 2k
Scroll down	LS ES K LS TA I LS 1k UA
Erase from beginning of line to cursor	LS LA 3p SH TA LA DE SH LA Q 4p

	SH LA A 6p
	SH LA Z 2p
	SH ES LA 5p
	SH OR LA 1p
Erase to end	LA Q 4p
	LA A 6p
	LA Z 2p
	ES LA 5p
	OR LA 1p
	TA LA DE
Delete line	ES OR K
	OR TA I
	OR 1k UA
	OR Q O
	OR A L
Insert line	SH ES OR K
	SH OR TA I
	SH OR 1k UA
	SH OR Q O
	SH OR A L
Set top left corner of window	Z A L
	Z ES K
	Z 1k UA
Set bottom right corner of window	SH Z A L
	SH Z ES K
	SH Z 1k UA

Figure 1. Keyboard Combinations for Special Screen Editor Functions

Calling The Monitor

Those who make heavy use of the built-in monitor can enter it with a SYS 54386. This mode of entry gives a "call" entry rather than the "break" entry you get with a SYS 1024. There are two observable differences between the two forms. A "call" entry gives a *C message on the screen, and does not change the value in the stack pointer (SP). A "break" entry gives a *B message, and decrements the value in the stack pointer by two. The "break" feature was not designed as the normal method for getting into the monitor, but rather as a tool for machine language programming. It's possible that, if you went back and forth from BASIC to monitor many times using the SYS 1024 "break" entry, you could run out of stack pointer space, although it's rather unlikely. Incidentally, for the older PET/CBM 2001-16 and -32, the "call" entry for the monitor is SYS 64785.

Automatic Program Adjustments

Many folks use location 50003 to allow a program to figure out what

kind of PET/CBM computer it's running on. PRINT PEEK (50003) gives a value of 0 on "old" PET's (version 1, BASIC 2.0), a value of 1 on "new" PET/CBM (version 2, BASIC 3.0), and now a value of 160 on the CBM 8032 (BASIC 4.0). Since many page zero locations in 8032 are the same as in the "new" PET/CBM's, some programs designed to run on either "old" or "new" versions can be adapted for the 8032 as shown in Figure 2.

Original program

```
10 PV=PEEK(50003)
20REM:=0 for OLD PETs,=1 for NEW
```

Modified for 8032

```
10 PV=PEEK(50003)
15 IF PV=160 THEN PV=1:?"Program running on CBM 8032"
20 REM:=0 for OLD,=1 for NEW,=160 for 8032
```

Figure 2.

Of course, this modification will not adapt all programs for the 8032. I've seen very few programs for 40-column machines whose output looks "right" on the 80-column unit (those which do are the ones without sophisticated graphics or formatting). If the program uses built-in routines from the PET/CBM ROM, it will take more effort to find the routine in the 8032 and modify the program to use it.

Hidden Memory

As in previous machines, the screen memory appears to "use up" memory addresses from 32768 to 36863, although only the first 2000 of those are "real" screen memory addresses. Another 2000 are "image" addresses, due to the incomplete decoding of those addresses. Of particular interest are the 48 addresses from 34768 (87D0 87FF) through 34815 which do not appear to be used for anything. That memory space could be used for short machine language routines, or data values that need to be tucked away where BASIC can't hurt them.

One bug I discovered in the 8032 is that a PRINT "[HOME]" often returns the cursor to the second line on the screen, rather than the first.

It was very frustrating to me to discover that many of the excellent machine language tools I've obtained via Jim Butterfield and Carl Moser do not work on the 8032. For those fortunate enough to have access to a 2040 disk drive, a 2001-32, and an 8032 all at the same time, it's possible to create a "host-target environment" or development system for the 8032.

Old Tools for New Programs

The 8032 and 2001-32 can both be connected to the 2040 using the IEEE-488 ports and the appropriate cables. A program "saved" to disk from one machine can be loaded into the other, and the transfer will work either way. You must be careful, however, not to have both computers trying to access the disk at the same time, or the system will get locked up. I've also experienced lockups when one of the computers is running certain machine language programs.

If you want to create an assembly language program for the 8032, you can use a good assembler (like the MAE from Eastern House Software) running on the 2001-32. After assembling the program in the 2001-32, use the built-in monitor to save the resulting machine language to disk. When the disk file is then loaded into the 8032, it will go into the memory locations corresponding to those from which it was saved.

Another thing I wanted to do was to look at the ROM in the 8032. Unfortunately, the only disassembler I had that would run on the 8032 was written in BASIC, and was exceedingly slow. On the other hand, I had several machine language disassemblers that were quite fast, but would not run on 8032. The solution was to copy a block of 8032 ROM, for example $B000 to $BFFF, into free RAM, say $1000 to $1FFF. This can be done in command mode with a statement like

```
FOR I=0 TO 4095:POKE 4096+I,PEEK
(45056+I):NEXT
```

When this is finished, we use the 8032's monitor to save the copy ($1000 to $1FFFF), which can then be loaded into the 2001-32 for examination. The choice of locations, obviously, must be such that it will not interfere with any of the tools being used to examine the code.

ROM Features

The monitor in the 8032 is very similar to that in the 2001-32, except for having been relocated. This is both good and bad. It's good because the 2001-32 monitor is documented, which allows us to figure out some of the ROM routine locations in the 8032 which correspond to known routine locations in the 2001-32. It's bad because there are many improvements which should have been made. It's a shame to waste half the screen, when we could be seeing twice as many locations on the 80 column machine. It's also a shame to have such limited capabilities in a monitor, when so many

good ones are available.

In the 8032, the operating system ROM starts at $B000 rather than $C000, which means there are only two free ROM sockets. Obviously, Word Pro 4 will take up at least one of those when it appears (this is being written in mid-August, and we've not been able to obtain a production copy yet).

Reader Feedback

We expect to be learning many more features of the 8032 in coming months, especially when we are able to get one of the new 8050 disk drives and test its interactions with old and new computers.

Any COMPUTE! readers who would like to contribute their discoveries may forward them to me, and I'll incorporate them in a future article (giving credit to the first contributor of each item). I would be especially interested in keyboard combinations that cause a shift from graphics to business mode (upper and lower case letters) and the ones to cause the screen to scroll up (without having to cursor down to the bottom of the screen).

Send your contributions directly to me at Virginia Micro Systems, Inc., 14415 Jefferson Davis Highway, Woodbridge, VA 22191.

Un-Crashing On Upgrade ROM Computers

Jim Butterfield

Here's the original "Uncrashing" article.

If you do much work in machine language, sooner or later you'll write a program that will crash.

Formerly, you were out of luck. Unless you were lucky enough to stumble into a type 1 crash — which would take you to the Machine Language Monitor, or to an ?INVALID NUMERIC statement — your only remedy would be to reset, and wipe memory.

Type 2 crashes (tight loops) could be guarded against with a little preparation involving fiddling with the interrupt structure. But the nasty type 3 crash (X2 codes) cannot be fixed without kicking the Reset line; and Reset means memory test, and memory test means you'll have to reload your program.

No more. On upgrade ROMs, you can come out of a hard crash with memory preserved.

Method: Set the diagnostic sense pin to ground, then kick the Reset line. The processor will re-awaken in the Machine Language Monitor with memory preserved.

There's more: you're not yet out of the woods. Type a semicolon followed by RETURN; PET will respond with a question mark. Now move the cursor back to your register display line, and change the Stack Pointer (SP) value from 01 to F8. This strange procedure is important: you must follow it exactly. Once you've done so, you're clear. You may return to BASIC with an X if you like, or proceed in the MLM.

Hardware: To make the diagnostic sense pin: take a standard 12-pin edge connector and wire pin 5 (diagnostic sense) to pin N (ground). Key the connector so it sits on the parallel user port. Plug it in whenever you want to un-crash, but don't leave it on the machine.

The Reset button is a little trickier, since you have to know where to connect it. Check with someone who's knowledgeable on PET hardware.

Commercial sources: International Technical Systems. Box 264, Woodbridge VA 22194 makes a Reset button.

Gord Reithmeier, 411 Duplex Avenue, Apt. 11, Toronto

Canada M4R 1V2, makes two uncrashing devices, either of which fits on the Parallel User Port; they include a diagnostic pin toggle switch and a Reset button. An IC clip snakes inside PET's cover to connect to the reset line. Instructions are included. The basic unit sells for $20; or for $30 the unit also includes the Poor Man's D/A converter.

Memory Partition of BASIC Workspace

Harvey B. Herman

A 6502 microprocessor can address a total of 65K bytes of memory (RAM plus ROM). The address space for BASIC programs (RAM) is necessarily restricted to less than that without resorting to hardware tricks. However, most BASIC programs do not take up anywhere near the maximum amount of reserved memory (32K bytes for the PET). Occasionally it would be useful to have several short, noninteracting BASIC programs in memory at the same time. For example, we use short programs to check student laboratory calculations (J. Chem. Ed., Vol. 55, p. 654 (1978)). When multiple laboratories are in process it would be simpler to LOAD a tape containing a number of programs and have each student run the program appropriate for his experiment.

One way to combine programs is to renumber and merge individual programs with a subsequent re-save of the combination. There are several disadvantages to this approach. It is important to keep line numbers separate in each program to be merged or you may not be able to delete or LIST parts of the program (unnerving at first). An ordinary LIST of the program will show frequently unrelated parts as one program (not aesthetically pleasing). The student user must remember to RUN with a line number specified for his chosen segment (or risk being hopelessly confused). Finally this approach will not allow placing utility programs (written in BASIC) in reserved areas of memory unless they are merged with every program (a formidable task).

Since I frequently use a number of short programs and have unused memory, I thought it would be helpful to partition the BASIC workspace for storage of individual programs. For example, an 8K PET (7167 bytes free) could have three 2K partitions under control of a 1K master program. It is possible to make other configurations as long as the total does not overrun the free memory available. If the partitioning is done properly the stored programs would not interact with each other. Each program would "think" it was in a 1K PET. (I actually owned a 2K PET once when I had a memory failure.) The master program would be in charge of adjusting the necessary pointers so a given program could be accessed when requested by the user.

Microsoft BASIC (for the PET and other microcomputers) uses pointers to subdivide free memory. The table summarizes important pointers (at least for this discussion) for both old and new PETs. The following material is for the old ROMs. It is not necessary to do any hex arithmetic to use the method I will describe. However, it does help to understand a little about pointers. If BASIC program text is stored beginning at location hex 0401 (it is assumed location hex 0400 contains a zero) the pointers to start of text (location 122/123) would read 1 and 4 for low and high byte respectively. That example was not too difficult but it must be remembered that the value returned is in decimal. If start of text was changed to, say hex 1001, location 123 would now read 16 corresponding to the decimal representation of the most significant half of that number (hex 10). To activate a new partition it is only necessary to set pointers to start of BASIC text (122/123), end of BASIC text (124/125) and top of memory (134/135). Subsequently executing CLR will set all the other pointers automatically (e.g., bottom of strings, etc.) and, after END, we find ourself in the new partition.

As an exercise I wrote a short master program (1K workspace) controlling three short donothing BASIC programs (each in a 2K workspace). They are shown in the figure. The master program asks the user for a program number and automatically sets the pointers to activate that program. At this point the user is in a 2K workspace with one program active which can be RUN or modified as desired. The last statement can be RUN or modified as desired. The last statement in each of the short programs returns the user to the master program. Each program is completely independent of the others, snug and protected in its own private world.

Setting up the example or one like it is not difficult. Each program could be typed in after the partition is activated by the master program (NEW first). Keep track of the size of each program by PEEKing at locations 124 and 125. This information should be stored in the master program so one can enter and leave the partition without destroying the BASIC text (c.f., line 210 in master program). The size of the master program should also be recorded and restored before returning to it (c.f., line 40 in program 1).

Relatively long programs are a nuisance to type into each partition. If the program is on cassette tape it can be relocated to any partition using the procedure described in my article "MOVE IT" (MICRO 16:17 and 17:18). Normally tapes load starting at hex 0400. By reading in the tape header first and changing the load parameters in the tape buffer, information on cassette tape can be

stored elsewhere in memory. Keep two points in mind. One, before using the relocated programs for the first time, the BASIC line links (see p. A-9 in PET User Manual) must be corrected. The easiest way to do this is to type any line number *not* in the program and return. Two, record the length of the program by PEEKing at locations 124 and 125 after an *ordinary* tape load. In my example program I showed 4 and 74 respectively. Since I intended to relocate the program to a partition beginning at hex 800, it was necessary to use the values 8 and 74 in line 210 in the master program.

The partition idea described above should be applicable, with only minor changes, to any microcomputer using Microsoft BASIC. In fact I used a partition for the first time on my SYM to store an initialization program which was used infrequently. In this case the partitions were of unequal length, 4K and 8K. Readers might be interested in storing their short BASIC utility programs in an out-of-the-way partition and activate the programs when necessary as I did with the SYM initialization program. Maybe others could share their ideas on the subject with me care of this magazine. We could publish the best ones in a future article. (Anyone for time sharing?)

Important PET Pointers (Low/High Bytes)

	ROM	Upgrade ROM
Start of Text	122/123	40/41
End of Text	124/125	42/43
Top of Memory	134/135	52/53

```
10 REM MEMORY PARTITION-MASTER PROGRAM
20 REM  EXAMPLE:
30 REM THREE PROGRAM WORKSPACES
35 REM CREATED AT:
40 REM HEX 0800-0FFF  PROGRAM 1
50 REM HEX 1000-17FF  PROGRAM 2
60 REM HEX 1800-1FFF  PROGRAM 3
65 POKE 123,04:POKE 122,01
66 POKE 125,06:POKE124,57:CLR
67 POKE2048,0:POKE4096,0:POKE6144,0
70 :
80 REM HARVEY B. HERMAN
90 :
95 PRINT "WHICH PROGRAM DO YOU WANT";
100 INPUT"(1-3)";N
110 ON N GOTO 200,300,400
200 POKE 123,08:POKE122,01
```

```
205 POKE135,24:POKE134,0
210 POKE 125,08:POKE 124,74:CLR:END
300 POKE 123,16:POKE122,01
305 POKE135,24:POKE134,0
310 POKE 125,16:POKE 124,74:CLR:END
400 POKE 123,24:POKE122,01
405 POKE135,32:POKE134,0
410 POKE 125,24:POKE 124,74:CLR:END
```

```
10 REM PROGRAM 1
20 A=1
30 PRINT A
40 POKE123,4
45 POKE124,57
50 POKE125,06
55 POKE135,8
60 CLR:END
```

```
10 REM PROGRAM 2
20 B=2
30 PRINT B
40 POKE123,4
45 POKE124,57
50 POKE125,06
55 POKE135,8
60 CLR:END
```

```
10 REM PROGRAM 3
20 C=3
30 PRINT C
40 POKE123,4
45 POKE124,57
50 POKE125,06
55 POKE135,8
60 CLR:END
```

The Deadly Linefeed

Jim Butterfield

Jim's advice in "The Deadly Line Feed" is definitely worth heeding. Just try it once the other way! Incidentally, this precaution is not necessary with BASIC 4.0.

When you write a BASIC statement like PRINT X, you print the value and start a new line.

To start a new line, the PET sends two characters: a RETURN, which terminates the old line, and a LINEFEED, which is often not needed and is sometimes deadly.

The linefeed character (CHR$(10)) is there to tell some types of printer that it's time to move the paper up. The Commodore printers don't need it, but others often do.

There are at least two cases, however, when you must not send the linefeed character — it will give you trouble.

Case one is when you're sending data to a disk file. If you should write this character to disk, you'll read it later — and it will give you problems.

Case two is when you're sending a formatted line to the Commodore printer — that is, to secondary address #1. It will seem to work in many cases; but you'll have problems when you try to change the format line by addressing secondary address #2.

How do you avoid sending the linefeed? Don't let PET terminate a line for you: do it yourself by sending the RETURN character.

So instead of sending PRINT#5, X code PRINT#5, X;CHR$(13); and be sure you don't forget the semicolon at the end of the line. If you have a lot of print lines of this type, you can set the RETURN into a string variable and save space: say R$=CHR$(13) and then you can code PRINT#5, X;R$; to do the job.

Using The GET Statement On The PET

Alfred J. Bruey

The GET statement is one of the least standard of any in Microsoft BASIC. Here's how you can use it on the PET, for programming or converting programs from one machine to another.

Although most programs use the INPUT statement to enter data during the execution of a BASIC program, the GET statement may be used to advantage, especially in cases where the program is to be run by a non-computer-oriented person.

The GET statement retrieves one character at a time from the keyboard buffer. Usually it is used with a string variable. The statement has the form

```
20 GET A$
```

This statement assigns the next character in the keyboard buffer to the string variable A$.

Since the GET statement executes as soon as it is encountered, whether there is anything in the keyboard buffer or not, it is usually necessary to check for the null character with a statement of the form

```
20 GET A$ : IF A$ = "" THEN 20
```

The IF . . . THEN statement puts the program in a loop until a key is pressed.

In the examples above, the value will not be displayed on the screen, so you will probably want to add a PRINT statement so you can see what key you pressed. The program will then look like this:

```
20 GET A$ : IF A$ = "" THEN 20
30 PRINT A$
```

The GET statement also does not prompt the user for input so we would normally add a PRINT statement before the GET statement:

```
10 PRINT "ENTER A CHARACTER";
20 GET A$ : IF A$ = "" THEN 20
40 PRINT A$
```

This short program prompts the user for a character, waits until he enters it, and then prints it on the screen.

If you try this example, you'll notice that you don't have to use

the RETURN key to enter the data. The ability of the GET statement to accept a character without the use of the RETURN key is a great advantage in a program where the user has to enter many one character answers, such as Y or N for YES or NO, or in a game where the progam has to recognize that you are depressing one of the cursor keys. For example, let's look at a program that waits for a Y or N answer:

```
5 REM WAIT FOR A Y OR N ANSWER
10 GET A$ : IF A$ <> "Y" AND
A$ <> "N" THEN 10
30 REM GO HERE IF Y OR N IS
ENTERED
```

For our next step, let's examine a short routine that will let us enter exactly five characters with a GET statement.

```
10 PRINT "ENTER FIVE
CHARACTERS";
20 FOR I = 1 TO 5
30 GET A$(I) : IF A$(I) = "" THEN 30
40 PRINT A$(I)
50 NEXT I
```

This routine allows any five characters to be entered. If you were to write a program for a non-programmer, you would probably want to write the program so that it would not recognize some of the special characters, such as the cursor-movement and clear-screen characters. Also, we would want to prohibit the user from entering a RETURN until he had entered the required number of characters. As a final touch, we will require that the user be able to use the DELETE key so he can correct errors during the keyboard entry.

As an example that uses the restrictions above, consider the following listing. This program allows you to enter nine numeric characters followed by the RETURN. The DELETE key is active during the process. This routine could be used to enter a social security number. First the listing, then a line-by-line explanation of the coding:

```
10 FOR I=1 TO 10
20 GET S$(I):IF S$(I)="" THEN 20
30 IF(S$(I)<"0" OR S$(I)>"9") AND ¬
      ¬S$(I)<>CHR$(20) AND S$(I)<>CHR$(13)
      ¬ THEN 20
40 IF S$(I)=CHR$(20) AND I=1 THEN 20
50 IF S$(I)=CHR$(20) THEN PRINT CHR$(157)
      ¬+" "+CHR$(157);:I=I-1:GOTO 20
54 IF I<10 AND S$(I)=CHR$(13) THEN 20
55 IF I=10 AND S$(I)<>CHR$(13) THEN 20
```

```
57 IF I=10 AND S$(I)=CHR$(13) THEN 70
60 PRINT S$(I);
70 NEXT I
80 END
```

Line 10: The beginning of the FOR . . . NEXT loop

Line 20: waits for a character to be entered

Line 30: A character gets rejected and execution returns to line 20 unless the character entered is one of the following:

a. an integer from 0 to 9
b. the DELETE character (CHR$(20))
c. the RETURN character (CHR$(13))

Line 40: Rejects the DELETE character if it is the first character, since there is nothing yet to delete.

Line 50: If it is a DELETE character, but not the first character, print a BACKSPACE, blank, BACKSPACE. Then reduce the character count by one and go back for another character.

Line 54: Reject RETURN if it isn't the tenth character

Line 57: Goes on with the program if tenth character is RETURN

Line 60: Prints character 0 to 9 on the screen

Line 70: End of FOR . . . NEXT loop.

It may look complicated, but the coding was necessary to protect the system against invalid entries. You also might want to disable the STOP key to keep the user from breaking out of the program.

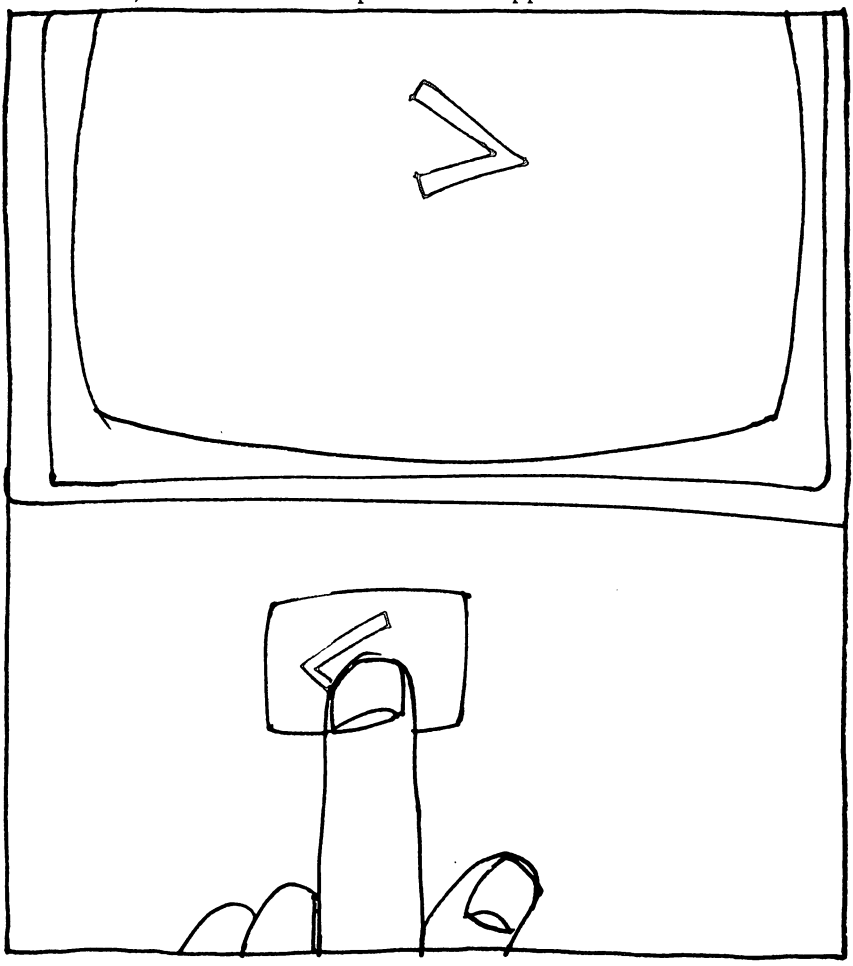
The preceding examples should get you started with the GET statement. I think you'll find it a desirable alternative to the INPUT statement in many of your programs.

Apparent Malfunction of the < Key

Jim Butterfield

In many PETs, the less-than (<) key will appear to be dead if cassette tape drive #1 is disconnected.

If you have to run without a tape drive, you might like to make up a plug for the tape edge connector. Putting a ground on the input line (connecting pins A-1 and D-4 on the cassette edge connector) should make the problem disappear.



Shift Work

Jim Butterfield

There are quite a few little tricks yet to be discovered on the PET. Here's a whole article on the Shift Key.

The SHIFT keys on the PET are pretty straightforward, right? Hold either one down while you hit another key, and you get the key's shifted equivalent: upper or lower case or a graphic. Not much to be said there.
Well, maybe one or two things . . .

Shifted Return

RETURN does two jobs: it takes you to the start of the next line, and it executes the line you're leaving.

Sometimes you don't want to execute the line. You're just drawing a picture on the screen. When you hit RETURN, the computer will take that part of the Klingon attack vessel you've just drawn and try to execute it as a BASIC command; you get ?SYNTAX ERROR, which doesn't help your picture much.

Other times, you have a BASIC line, but you don't want to execute it yet. Maybe you've got a little muddled up with the programmed cursor and every time you try to back up the cursor to fix things you get another unwanted graphic. You don't want to press RETURN and enter this botched line into your program before you have a chance to fix things up.

Just hold down SHIFT as you press RETURN and you'll go to the next line without trying to execute what you've just done.

Shifted Space

When you press SPACE, the PET prints a space. When you hold down the SHIFT key and press SPACE, the PET prints a space. Same thing. The shifted SPACE, however, is a different character on the screen. Looks the same, but it's not a true space. How can you use this? Here's one very handy application. Suppose you want to do an INPUT and don't want the user to accidentally stop the program by typing RETURN without input. Shifted-space will do the trick. Try this tiny program:

```
10 INPUT "(see note below)";X$
20 PRINT "THANK YOU.":GOTO 10
```

Here's what to put between the quotes on line 10. After you type the first quote mark, hold down the shift; type three spaces; type

three cursor left characters. Now release the shift and complete the line, starting at the second quote mark.

I call this program ABUSE. After a hard day at the computer, you can put this one in, and proceed to call it every name under the sun. It will thank you and ask for the next insult.

The interesting thing is that the program won't stop if you press RETURN without input. That invisible shifted-space that you have printed to the right of the question mark is a genuine input character. If you don't write over it with your own information, it will be accepted as input, and the program won't stop. Instead, it will humbly thank you . . . for nothing.

Pseudo-shifted Characters

There's a group of characters that you can't input via keyboard/screen, but which are useful in certain types of file handling. The shifted RETURN is also useful in this application.

Here's the problem. When you use the INPUT# statement for receiving data from tape or disk, the input procedure stops on three characters: comma, colon, and RETURN.

This is annoying when you're trying to input names from an address list like DR. ALOYSIUS CHIP, PHD or HORACE SCHMEDLAMP, JR. or have address lines like ATTENTION: MURPHY. The input routine nearly drops the PHD, JR. and MURPHY, and you're left to scratch your head over why the data disappeared.

Relief is in sight. If you can catch the comma or colon before you write it to file, just change it to its shifted equivalent by adding 64 decimal to the ASCII value. It takes a little more work when you write it, but it saves work and puzzlement when you read it back later. The general technique for a single character is:

```
A = ASC(A$) : IF A=44 OR A=58 THEN A$=CHR$(A+64)
```

You can write the re-formed A$ to the file and feel secure that it will come back without trouble and print correctly.

Exactly the same thing can be done with quotation marks. The input routine assumes that ordinary quote marks are there so that they can be removed before you see the string. There's a good reason for this, but it doesn't help you when you really want them to be there and part of the input data. Once again, shift the quote by adding 64 to it. Since quotes are often directly program-generated (rather than input), you can just use CHR$ (98) instead of CHR$ (22).

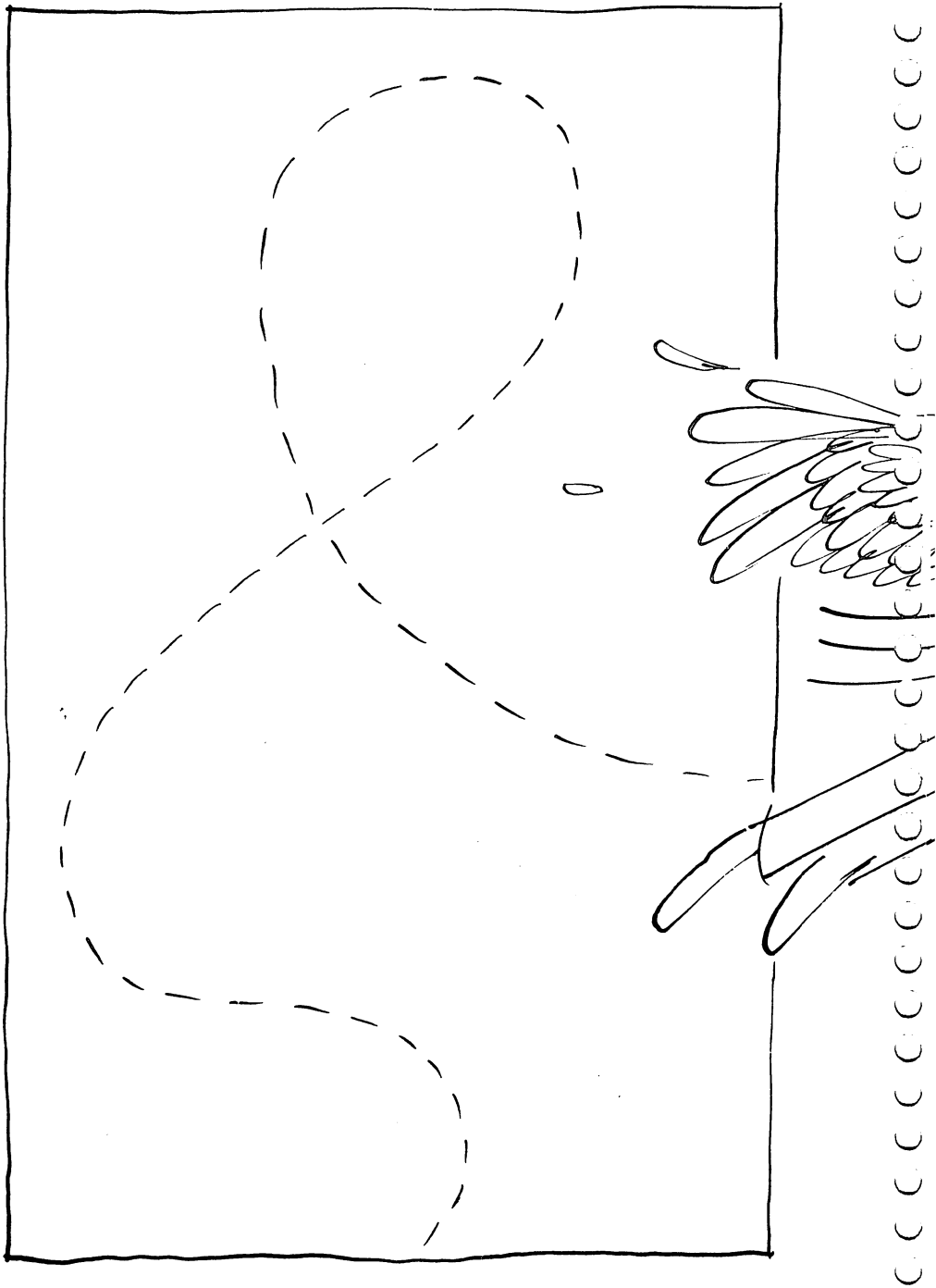
Occasionally, you may want to input two lines at a time from a

file. The shifted-return will do the trick. Oddly enough, you must add 128 to the RETURN to make a shifted return: it's CHR$(141). This always works great if your input goes to the screen. If you're using a printer, however, check it out to make sure it recognizes the shifted-return and does the right thing.

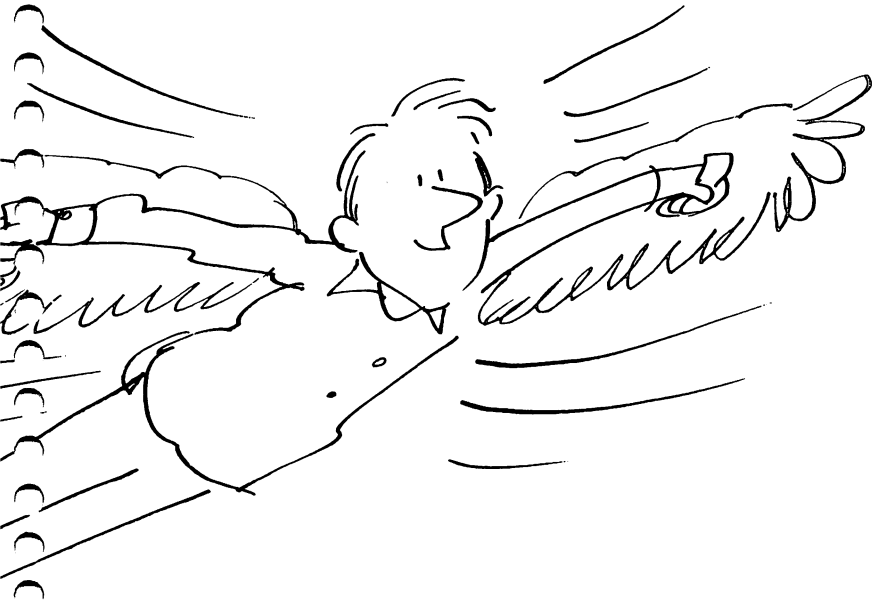
Afterthought

Before I leave you to shift for yourself, try this last little keyboard curiosity. Hold down both shift keys. Now, with your third hand, or nose, or whatever, try pressing a few keys on the left-hand side of the keyboard: Q, A, S, Z.

I don't know why you get the odd characters. There's probably a moral here: two shifts are not better than one? Too many shifts spoil the keyboard?



CHAPTER THREE: Beyond The BASICS



Mixing BASIC and Machine Language

Jim Butterfield

This is a logical extension of "Fitting Machine Language Into The PET." The example programs are useful, and illustrate the concept.

It's not too hard to put BASIC and machine language together. Care is needed, of course, but there's no great mystery.

One of the easiest tricks is to put the machine language program behind the BASIC program in memory. Once you've created and saved the package, it may be LOADed and SAVEed without special instructions. There's one thing you need to watch, however: when the package is complete, you must not change the BASIC program. If you do, the machine language part will be moved away from its original location. Your SYS command will take you to the wrong place, and the machine language program probably won't work anyway.

The following sample programs use this kind of packaging. Here's how to set them up on your machine:

1. Type in the BASIC program completely. Check it carefully, since you won't have the option of changing it later.
2. Enter the machine language monitor. If you happen to have an early PET with original ROM (no built-in monitor), you should have previously loaded one of the "high-monitors" that are available. (See Roy Busdiecker's article: "Relocate PET Monitor Almost Anywhere".)
3. Double check to ensure that your BASIC program hasn't somehow crept up above the machine language area that you plan to use. There are several ways to do this. One is to inspect the BASIC memory area and spot the three 00 values that signal the end of the BASIC program. Another way is to take a look at the start-of-variables pointer (hex 7C and 7D on original ROMs; hex 2A and 2B on upgrade ROMs) and make sure it's below the area you are about to work in.
4. Now type in the machine language as shown. Check it closely; a single mistake will cause improper operation.
5. Finally — still in the Machine Language Monitor — save the whole thing from start-of-BASIC (hex 0400) to end-of-machine-

language-plus one. On the Universal ROM Test program, for example, you'd save from 0400 to 084A.

Now your program is ready. It can be loaded, saved or copied without any special knowhow. Just remember — don't change the BASIC part of the program.

Universal ROM Test

This program loads into any machine and tells you what kind of ROM you have. It will test ROM repeatedly until you stop it . . . this makes it good for spotting intermittent errors.

All of the standard ROM sets I know about are there. There are also a couple of experimental Commodore ROM sets included — I had a chance to take a look at them during a recent trade show. These may change by the time Commodore releases them, so don't take them too literally.

```
100 PRINT"ñ↓↓ UNIVERSAL ROM TEST    JIM ¬
    ¬BUTTERFIELD"
110 DATA "Ø11 ORIGINAL",59487,12796,
    ¬51858,61980,58622,7753,6792,-1,-1
120 DATA "Ø19 ORIGINAL",59339,12796,
    ¬51858,61980,58622,7753,6792,-1,-1
130 DATA "UPGRADE PERSONAL",41799,42993,
    ¬64959,8803,38129,43129,23093,-1,-1
140 DATA "UPGRADE BUSINESS",41799,42993,
    ¬64959,8803,38129,43129,23093,-1,-1
160 DATA "DOS PERSONAL I",40596,4520l,
    ¬34900,08207,47820,00390,44555,
    ¬40847,44239
170 DATA "DOS BUSINESS I",40596,4520l,
    ¬27250,08207,47820,00390,44555,
    ¬40847,44239
180 DATA "DOS 80-CHARS I",40596,4520l,
    ¬21130,08207,47820,00390,44555,
    ¬40847,44239
190 DATA "*"
200 DATA 192,208,224,240,200,216,248,
    ¬176,184
210 DIMA$(8),V(8,9),A(9),R(9),M(9)
220 READX$:IFX$="*"GOTO290
230 R=R+1:A$(R)=X$
240 FORJ=1TO9:READV(R,J):NEXTJ:GOTO220
290 FORJ=1TO9:READA(J):NEXTJ
300 FORJ=1TO9:POKE1023,A(J):SYS2112
310 R(J)=PEEK(1021)+PEEK(1022)*256
320 NEXTJ:P=7
```

```
330 FORJ=1TOR:M(J)=0:FORK=1TOP:IFR(K)=V(
      ¬J,K)THENM(J)=M(J)+1
340 NEXTK,J
350 L=-1:FORJ=1TOP:IFM(J)>LTHENL=M(J):
      ¬K=J
360 NEXTJ:IFP=7ANDV(K,8)>=0THENP=9:
      ¬GOTO330
370 N=N+1:PRINT"hTEST";N:PRINT"↓↓↓↓";
      ¬A$(K):PRINT
380 FORJ=1TOP
390 IFR(J)=V(K,J)THENPRINT:GOTO410
400 A=A(J):B%=A/16:C=A-B%*16:PRINTCHR$(B
      ¬%+55);CHR$(C+48)
410 NEXTJ:PRINT
420 PRINT"BAD ROMS:";P-L;"←  ":GOTO300
.
.:  0840 AD FF 03 85 B2 A9 08 85
.:  0848 B5 A9 00 85 B1 85 B3 A0
.:  0850 00 18 71 B1 90 02 E6 B3
.:  0858 C8 D0 F6 E6 B2 C6 B5 D0
.:  0860 F0 8D FD 03 A5 B3 8D FE
.:  0868 03 60 00 00 FF 00 00 FF
```

RAM Test

This is a very fast memory test, yet it's quite thorough. It's adapted from the memory test in The First Book of KIM — you can dig out more details there if you're curious.

The coding is a little crowded; I wanted to fit the whole thing into 256 bytes so that the rest of memory would be available for testing.

The program tests memory repeatedly until stopped. Users with a full 32K of memory can input a value of 33 and test screen memory too. That way, they can see the actual test on the screen as it happens.

```
10 INPUT"ĥ↓↓HOW MANY K";K:K=K*4-1
20 POKE185,K:POKE184,5
30 N=N+1:SYS1156:PRINT"hTEST ";N;
40 J=PEEK(187):IFJ>KGOTO30
50 PRINT"FAILED AT";J*256+PEEK(186)
.
.:  0484 A9 00 A8 85 BA 85 BC A2
.:  048C 02 86 BD A5 B8 85 BB A6
.:  0494 B9 A5 BC 49 FF 85 BE 91
.:  049C BA C8 D0 FB E6 BB E4 BB
.:  04A4 B0 F5 A6 BD A5 B8 85 BB
```

```
.:  04AC A5 BC CA 10 04 A2 02 91
.:  04B4 BA C8 D0 F6 E6 BB A5 B9
.:  04BC C5 BB B0 EC A5 B8 85 BB
.:  04C4 A6 BD A5 BE CA 10 04 A2
.:  04CC 02 A5 BC D1 BA D0 15 C8
.:  04D4 D0 F0 E6 BB A5 B9 C5 BB
.:  04DC B0 E8 C6 BD 10 AD A5 BC
.:  04E4 49 FF 30 A1 84 BA 60 32
```

Tape Test

The version given is for Upgrade ROM only. This lets you watch any PET tape and see the kind of signals that are coming in from it.

I had hoped that this program would solve head alignment problems once and for all. It doesn't quite make the grade, since in my opinion it's not sufficiently sensitive to slight alignment changes. Even so, you will find the program instructive.

```
 100 PRINT"ñTAPE TEST # JIM BUTTERFIELD"
 110 POKE59468,12
 120 PRINT:X$="LEADER":GOSUB500
 130 X$="DATA":GOSUB500
 140 X$="ERROR":GOSUB500
 150 INPUT"TAPE UNIT";T
 160 IFT>2ORT<1GOTO150
 170 POKE212,T
 180 SYS(1280):END
 500 PRINT" UCCI"
 510 PRINT" B  H ~ ";X$
 520 PRINT" JFFK"
 530 RETURN
READY.
.
.:  0500 20 12 F8 78 A6 D4 CA F0
.:  0508 15 CE 13 E8 A9 90 8D 4E
.:  0510 E8 AD 40 E8 8E FA 00 29
.:  0518 EF 8D 40 E8 10 0B EE 11
.:  0520 E8 A9 34 8D 13 E8 8D F9
.:  0528 00 A9 6E 8D 90 00 A9 05
.:  0530 8D 91 00 58 20 F0 F8 2C
.:  0538 13 E8 10 F8 A2 02 A0 00
.:  0540 A9 20 95 B8 B5 B1 F0 06
.:  0548 94 B1 A9 A0 95 B8 CA 10
.:  0550 EF A5 B8 8D 7A 80 A5 B9
.:  0558 8D F2 80 AD 6A 81 06 BA
.:  0560 69 00 29 1F 8D 6A 81 20
.:  0568 29 F7 10 C7 30 C5 20 7A
```

```
.:  0570 05 2C 40 E8 2C 10 E8 4C
.:  0578 E4 E6 AE 49 E8 AD 48 E8
.:  0580 EC 49 E8 D0 F5 A0 FF 8C
.:  0588 48 E8 8C 49 E8 E0 FC 90
.:  0590 08 E0 FF D0 07 C9 50 90
.:  0598 0B E6 B3 60 E0 FE D0 10
.:  05A0 C9 60 90 0C A5 CC 29 FC
.:  05A8 F0 03 E6 B1 60 E6 CC 60
.:  05B0 A9 00 85 CC E6 B2 60 00
```

Leader Write

This is for Upgrade ROM only. It writes continuous "leader" (sometimes called "shorts") to tape. It's useful, in conjunction with Tape Test, in checking out various brands of tape for data quality.

```
 100 PRINT"ñ↓  ⌜ WRITE LEADER TAPE "
 110 PRINT"↓ #   JIM BUTTERFIELD"
 120 PRINT"↓ THIS PROGRAM WRITES A ¬
       ¬CASSETTE TAPE"
 130 PRINT"WITH 'LEADER' SIGNAL."
 140 PRINT"↓ THE CASSETTE TAPE SO PRODUCED
       ¬MAYBE"
 150 PRINT"USED WITH 'TAPE TEST' TO EITHER:
       ¬"
 160 PRINT"  --CERTIFY THE TAPE AS OK;"
 170 PRINT"  --ALIGN TAPE HEADS OF THE ¬
       ¬OTHER CASSETTE"
 180 PRINT"     UNITS. IN THIS CASE,
       ¬ BE SURE"
 190 PRINT"     THAT YOU ARE WRITING ON A"
 200 PRINT"     PRECISELY ALIGNED TAPE ¬
       ¬UNIT.↓"
 210 SYS1472:END
READY.
.
.:  05C0 A9 01 85 D4 20 47 F8 A9
.:  05C8 70 8D C3 00 78 A9 A0 8D
.:  05D0 4E E8 A2 08 20 9B FC A9
.:  05D8 02 85 DE A9 34 8D 13 E8
.:  05E0 8D F9 00 8D 49 E8 58 A9
.:  05E8 70 8D C3 00 20 35 F8 F0
.:  05F0 F6 20 7B FC 4C 84 F2 AA
```

Simulated BASIC In Machine Language

Blaine D. Standage

Would you like to be able to execute BASIC statements from within machine language programs? Here's how . . . (The $C702 entry point is for Upgrade ROMs. The 4.0 equivalent is $B787. For original ROMs it is $C6F5.)

There are probably very few among us who, having ventured into machine language programming, have not found themselves struggling long and hard to do something as simple as opening or closing a file. — "Its so easy in BASIC! Now I should be able to set these zero page values and JSR to $ . . . " — "Darn! Well, maybe if I . . ." If that stirs unpleasant memories, this article is just for you. I have been there many times. Often those hours of hard work produce a good result which I carefully record for future use (at which time it still may not work because of a subtle difference in the situation). And then there are times when I give up and write a companion program in BASIC to do only that one elusive task: "OPEN4,4: SYS4096: CLOSE4: END".

Now, I have two programs to do the job of one, two loads, coordinated revisions, etc. What a mess!

Recently, in a fit of frustration, I decided that if the PET could execute a BASIC instruction in a BASIC program, then there must be some way to make it do the same thing in a machine language program. And not just file related commands, nor even most of the commands. I wanted *all* the commands. Each and every one! After all, who's in charge here? Me or the PET?

In much less than the time spent on several previous individual problems, I had a general solution. I can now execute any BASIC command from within a machine language program with what I call "simulated BASIC" and the procedure is beautifully simple. So simple that it has become a double edged blessing/curse. The blessing is that I will never again be stuck fighting an isolated machine language problem that I could easily solve in BASIC unless I choose to. The curse is that, having such an easy way out, I may not struggle as hard as I should to learn more secrets of the ROM code.

For pure machine language programs the most popular use of

simulated BASIC will probably be the execution of IEEE bus and file related commands such as OPEN, CMD, SAVE, LOAD, and CLOSE. Complex arithmetic calculations are also a good application. How would you like to try "B=543*SQR(SIN(A))" in machine language? No problem using simulated BASIC! Just "simulate" the same instructions that you would execute if the calculation were being done in a comparison BASIC program. Naturally this must include the PEEKs and POKEs necessary to transfer data access to and from the machine language program.

In addition, there are some very interesting possibilities for BASIC programs with machine language subroutines. From within the machine language subroutine you might want to do something simple like changing the value of a BASIC variable without returning to the BASIC part of the program. Or, you might want to try something complex like making a multi-way branching decision to take you back to an instruction in BASIC other than the one you came from. Just be sure you understand the operation of the stack before you try branching operations with simulated BASIC. Branching is a little tricky, but it can be done.

Now that you have some idea of what simulated BASIC can do for you, let's seriously examine the way to make it happen.

Procedure

1. Create a string of bytes which look like a normal BASIC language instruction in memory.
2. Save the contents of the BASIC buffer pointer ($77 and $78), and the most significant byte of the current BASIC line number ($37).
3. Put the ADDRESS of the first byte of the "instruction" in the BASIC buffer pointer, and set $37 to zero.
4. Put the first byte of the "instruction" in the accumulator, set the carry bit, and JSR to ROM location $C702 where the "instruction" will be executed.
5. Restore the original values to the locations you saved in step 2.

The reasons for some points in the above procedure may not be obvious to all of us so let's dig a little deeper.

The simulated BASIC instruction created in Step 1 differs very slightly from a real BASIC instruction line.

The image of a real BASIC instruction in memory contains a next line address pointer (two bytes), a line # (two bytes), the instructions bytes, and a terminator byte. Since we are not truly

operating in BASIC, the next line address and the line number are not needed.

Within a real BASIC instruction, all commands, functions, and relational operators are saved in memory as tokens. This convention must be followed exactly in creating the simulated instruction.

For simulated BASIC the instruction terminator should be a $00 byte because multiple instructions in a single line are not allowed.

If you are using an assembler, the source code for an example instruction such as "OPEN9,4,0" might be:

```
.BYTE $9F,'9,4,0', $00
```

where $9F is the token for "OPEN". After assembly, the bytes in memory would be:

```
9F 39 2C 34 2C 30 00
```

In Step 2, saving the contents of the BASIC buffer pointer is optional, depending on how your program is structured. They need not be saved if your program is totally machine language and if it terminates in a "warm start" or other action which forces the zero page pointers to a known state.

If you have any doubts, play it safe and save the pointers!

There are a few instructions such as INPUT and GET which can be executed only in "program" mode and the ROM code performs two tests to determine the current mode. One of these tests involves the BASIC buffer pointer bytes and, since we are already altering them, this test will pass. The other test checks the contents of location $37, which is the most significant byte of the current BASIC line number. To execute these special instructions in simulated BASIC, location $37 must be set to some value other than $FF. Since changing this byte does not affect the execution of the other instructions, I usually set it to zero and restore it (see Step 5) as a standard part of the procedure.

Steps 3, 4, and 5 are straightforward. Just remember that the least significant byte of the "instruction" ADDRESS goes in location $77 and the most significant in $78, and don't forget to set the carry bit or you'll get some very odd results.

That's all there is to it. The whole world of BASIC is now available to you in machine language. Program what you can or what you choose to in machine language and do the things you find difficult in simulated BASIC.

Implementation

Now I would like to suggest a method I have found most useful in implementing simulated BASIC. Table 1 contains source code, slightly modified for clarity, from an operational program. Depending on which assembler you have, there may be minor differences between this source code and the code that you will actually use, but the methods illustrated are applicable in any case.

If you don't have an assembler, don't be discouraged. All the advantages of simulated basic are still available to you. Its just going to be more difficult for you to get the code into your computer, but you should be used to that by now.

The first six lines in the table show code which would be contained in the body of your program. They cause the execution of the three simulated BASIC instructions which you will see near the end of the table. The values which are loaded into the X register are the number of bytes between the start of the "instruction" table and the start of the individual "instruction" that is to be executed.

The code in the center section of the table is a subroutine which performs the execution steps which were described previously (see Steps 2 through 5).

An advantage of this subroutine method over others I have tried is that the contents of the "instruction" list can be changed or added to without any manual re-calculation of X register index values. Those values are computed at assembly time as a result of the last three source code lines in the table.

General Comments

The key location in ROM ($C702) for simulated basic execution is for a 2001 series PET with Upgrade ROM. *(See Editor's Note at start of article for other machines).*

Next, every BASIC instruction I have executed using "simulated BASIC" has worked exactly as expected, but I obviously have not tried anything approaching all the possible applications, so, don't be surprised if, somewhere along the old flow chart, something unusual should happen. Just be sure to let us all know about it.

Finally, for those of you who may want to double check your construction of a "simulated BASIC" code line: Simply NEW the computer, type in the desired line of code, (complete with line number) then SYS 1024 to the monitor and look at the code from location $0405 to the first "zero" byte.

Table 1.

SAMPLE IMPLEMENTATION OF SIMULATED BASIC

```
7043  A2  00          LDX #INDEX1
7045  20  7F  72      JSR BEXEC
7048  A2  07          LDX#INDEX2
704A  20  7F  72      JSR BEXEC
704D  A2  10          LDX #INDEX3
704F  20  7F  72      JSR BEXEC
7052  60              RTS OR MORE PROGRAM
                ;
                ;SAVE BASIC INSTRUCTION POINTER
                ;AND LINE# MS BYTE
727F  A5  77    BEXEC LDA $77
7281  8D  4F  77      STA $774F
7284  A5  78          LDA $78
7286  8D  50  77      STA $7750
7289  A5  37          LDA $37
728B  8D  51  77      STA 7751
                ;
                ;SET NEW POINTER & LINE# VALUES
728E  18              CLC
728F  8A              TXA
7290  69  B3          ADC #<INSTR1
7292  85  77          STA $77
7294  A9  00          LDA #$00
7296  85  37          STA $37
7298  69  72          ADC #>INSTR1
729A  85  78          STA $78
                ;
                ;FIRST INSTRUCTION BYTE TO ACC.,
                ;SET CARRY BIT, AND EXECUTE
729C  BD  B3  72      LDA INSTR1,X
729F  38              SEC
72A0  20  02  C7      JSR $C702 (B787)
                ;
                ;RESTORE BASIC POINTER & LINE#
72A3  AD  4F  77      LDA $774F
72A6  85  77          STA $77
72A8  AD  50  77      LDA $7750
72AB  85  78          STA $78
72AD  AD  51  77      LDA $7751
72B0  85  37          STA $37
```

```
72B2  60                    RTS
                      ;
                      ;
                      ;'BASIC' INSTRUCTION TABLE
                      ;OPEN6,4,0
72B3  9F  36  2C      INSTR1 .BYTE $9F, '6,4,0',$00
72B6  34  2C  30
72B9  00
                      ;PRINT#6 CHR$(13)
72BA  98  36  2C      INSTR2 .BYTE $98,'6,',$C7,'(13)',$00
72BD  C7  28  31
72C0  33  29  00
                      ;CLOSE6
72C3  A0  36  00      INSTR3 .BYTE $A0,'6',$00
                      ;
                      INDEX1=INSTR1-INSTR1    72B3-72B3=0
                      INDEX2=INSTR2-INSTR1    72BA-72B3=7
                      INDEX3=INSTR3-INSTR1    72C3-72B3=16
```

Fitting Machine Language Into The PET

Jim Butterfield

Machine language needs an area of protected memory.

A PET machine language program must co-exist with BASIC. You need at least one BASIC instruction, even if it's only a SYS command to start the machine language running. You could give the SYS as a direct command from the keyboard; but it's usually much better to run it in as a program line and let the user type RUN.

The BASIC program is usually in a predictable place. It will start at address 1025 (hexadecimal 0401), and will occupy memory space upward from there. The end of the BASIC program can be spotted by the fact that memory will contain three consecutive zeros.

Your machine language program can go almost anywhere that's free. There are three favorite places for such programs:

— in the cassette buffer(s);
— immediately above the end of BASIC;
— at the top of memory.

Each has its advantages and drawbacks. Let's deal with them one at a time.

Cassette Buffer(s)

If you use only cassette number 1, the second cassette buffer is free and available for your use. Its address is hex 033A to 03F9, which gives you 192 locations to play with. If you don't use cassette tape at all, i.e., you use disk, you may use both buffers; this gives you addresses from 027A to 03F9 hex, or 384 locations.

The newest models of PET/CBMs use a small portion of the second cassette buffer. If you have 4.0 ROMs — that is, your machine accepts English-language commands like SCRATCH or CATALOG — you'll need to leave the bottom twenty locations or so free. And if your machine has a TAB key, you must leave a few locations at the top — or your tab stops will change mysteriously. On these newer models, work in the range of 0350 to 03ED and you'll be reasonably safe. (848) (1005)

The cassette buffer area is ideal for machine language

programs. Except as noted above, it is completely unused for any BASIC activity. Loading new BASIC programs won't affect it. You may easily save machine language and BASIC together by using the machine language monitor's .S (Save) command and specifying the address range, from start-of-machine-language to end-of-BASIC. The whole thing will be saved; and later, a LOAD from BASIC will load everything back, both machine language and BASIC.

There are two problems. First, the amount of space is limited. You'll find plenty of room for your first small programs, but as you get more experienced and more ambitious your programs will become too big to fit into this space.

The second problem may not be a problem at all, depending on your objectives. Programs which have been written and saved using the above techniques can't be copied easily. The naive user who performs LOAD and then SAVE will load the whole program — but will save only the BASIC part. To copy the program, you need to go to the machine language monitor — and then you need to know the addresses to give for the .S (Save) command. If you prefer to keep your programs private, this can be a useful technique. But if you'd rather see them passed around, this can be a problem — you might get tired of being the only person that can make copies for other people.

Above The End Of BASIC.

This isn't hard to do, once you get the hang of adjusting the start-of-variables pointer (located at 2A and 2B hex in upgrade ROM, or at 7C and 7D in original ROM). After your BASIC program is written, this pointer will direct you to the available memory immediately after that program. Put your machine language program there, and then change the pointer so that it indicates a location above both programs — both BASIC and machine language. Give the BASIC CLR command after you do this, and all the other pointers will line up correctly.

Now you can use a BASIC SAVE to record your program. Both BASIC and machine language will be saved, and they will both load together at a later time.

Using this system, you'll have lots of space for your machine language program if you need it. The composite program can be copied easily: just do a conventional LOAD and SAVE any time you want an extra copy.

There's one drawback to using this system. Once you have set it up, it is difficult to make a change to the BASIC program. If you

add or delete anything — even a single character — your machine language program will move to a new location. You'll need to change your SYS command or USR vector. Worse, most machine language programs can't be moved without needing changes. You will have to rewrite the program so that it will work properly in its new location.

This is one of the most convenient ways to position machine language. Keep in mind, however, that it reduces your freedom to change the BASIC program.

At The Top Of Memory

This is a very convenient place to put relatively permanent machine language programs. It goes in the high end of memory. Since string variables are written in high memory, you must protect this type of program by moving the limit-of-BASIC-memory pointer down so that it points below your program. (This pointer is located at hexadecimal 34 and 35 in upgrade ROM, or hex 86 and 87 in original ROM). Once you have done so, the program will take up permanent residence and will usually remain in your machine until you turn the power off.

In this case, the BASIC and machine language programs are no longer adjacent in memory. You can't SAVE and LOAD them together. Be sure to LOAD the machine language program first, followed by the BASIC program; otherwise some of the BASIC pointers will be mixed up, and you'll probably get an ?OUT OF MEMORY error. Remember, too, that the machine language program will take up permanent residence. It won't go away when you load a new BASIC program.

Another problem that you'll have to face with this technique is that different machines are fitted with different amounts of memory. A machine language program that sits neatly at the top of an 8K machine will lie smack in the middle of memory on a 16K unit. You would need different versions for different sizes of machine. Keep in mind, once again, that you usually can't move a machine language program to a new location without making changes to it.

It is possible to write a program which finds the top of memory, parks the machine language program up there (wherever it happens to be), makes all necessary corrections, and then moves the limit-of-BASIC-memory pointer to the proper place. Many programs, such as Superman and the DOS "wedge" system, do exactly that. It's an advanced technique, however — don't try your hand at it until you feel you're ready.

Machine Language Code For Appending Disk Files

Robert H. Wollenberg

A common problem (appending or merging programs) has been solved in several ways, but many users still don't have this capability. Mr. Wollenberg's program, written for the upgrade ROM PET, elegantly does the trick.

The attached machine language routine provides a very useful tool for those who have been frustrated (as I have) by the inability of the current disk operating system (DOS) of the Commodore 2040 Dual Drive Disk to append programs. Although firmware recently introduced by Palo Alto IC's (The Programmer's Toolkit) provides some relief to those who require a convenient appending procedure, it suffers a serious drawback. The system operates only by appending a tape file to a program already in memory. Thus, it becomes necessary to first save program pieces to tape and then reload these in proper sequence using the append command. Since I was reluctant to use this slow tape file procedure, I searched for alternatives involving disk files.

A little investigation of the DOS command, Copy, reveals that ths instruction goes a long way toward solving the problem. BASIC is stored in memory starting at location $0401. The first two bytes are forward pointers to the next line of code (stored by the usual 6502 convention of low byte/high byte). Locations $0403 and $0404 store the line number for this first line of BASIC. The ASCII Code for this line is stored beginning at location $0405 and ends with the delimiter zero. The next byte begins the second line of BASIC and is stored at the location pointed to by the forward pointers at locations $0401 - $0402 described above. By following the forward pointers from line to line, one eventually reaches the end of the BASIC code. This event is signaled when the forward pointers are zero.

When the DOS Copy command concatenates two disk files, the zero page values indicate that both programs were combined; however, listing the concatenated program reveals only the first program. The second program, for all purposes, remains invisible to the BASIC interpreter. This is because the forward pointers of the first program eventually point to the two delimiters, zero, and at

```
0015  0000                    * = $033A

0017  033A  A9 FF       START  LDA #$FF        ;SETUP TO FIND PROGRAM DELIMITER
0018  033C  85 11              STA $11
0019  033E  85 12              STA $12
0020  0340  20 2C C5           JSR $C52C       ;ROM FIND LINE # ROUTINE
0021  0343  20 6C 03           JSR ENDBAS      ;CHECK FOR END OF BASIC
0022  0346  F0 21              BEQ DONE        ;YES, EXIT AND PRINT READY
0023  0348  A5 5C              LDA $5C         ;CURRENT LINE
0024  034A  D0 02              BNE SKIP
0025  034C  C6 5D              DEC $5D         ;MOVE BACK ONE BYTE
0026  034E  C6 5C       SKIP   DEC $5C
0027  0350  A9 00              LDA #$00
0028  0352  A0 04              LDY #$04        ;SETUP FOR 4 LOOPS
0029  0354  91 5C              STA ($5C),Y     ;STORE NEW PROGRAM DELIMITER
0030  0356  A9 20              LDA #$20        ;LOAD ASCII BLANK
0031  0358  88                 DEY
0032  0359  91 5C       AGAIN  STA ($5C),Y     ;STORE BLANKS
0033  035B  88                 DEY
0034  035C  10 FB              BPL AGAIN
0035  035E  20 72 C5           JSR $C572       ;ROM 'CLR' ROUTINE
0036  0361  20 42 C4           JSR $C442       ;ROM FIX CHAINING ROUTINE
0037  0364  20 6C 03           JSR ENDBAS      ;CHECK FOR END OF BASIC
0038  0367  D0 D1              BNE START       ;DO NEXT PROGRAM IF MORE
0039  0369  4C 8B C3    DONE   JMP $C38B       ; EXIT TO BASIC
0040  036C  38          ENDBAS SEC             ;CHECK FOR END OF BASIC ROUTINE
0041  036D  A5 2A              LDA $2A         ;TOP OF VARIABLES
0042  036F  E5 5C              SBC $5C         ;CURRENT LOCATION
0043  0371  C9 02              CMP #$02        ;DIFFERENCE = 2 IF END
0044  0373  D0 04              BNE NOTEND      ;RETURN IF NOT DONE
0045  0375  A5 2B              LDA $2B         ;CHECK HIGH ADDRESS
0046  0377  C5 5D              CMP $5D
0047  0379  60          NOTEND RTS
```

```
 0 REM ROBERT H WOLLENBERG
 10 REM PROGRAM TO APPEND USING DISK CONCATENATE COMMAND
 15 DEF FNA(X)=PEEK(X)+256*PEEK(X+1):DEF FNB(X)=FNA(X)-X
 20 I=FNA(40):E=FNA(42):X=FNB(I)
 25 IF I+X>=E-2 THEN PRINT"END OF MEMORY-NO LINK":END
 30 I=I+X:IF FNA(I)<>0 THEN X=FNB(I):GOTO 25
 35 FOR J=I TO I+3:POKEJ,PEEK(J+4):NEXT
 40 FORJ=I+4 TO I+7:POKEJ,32:NEXT: Y=I-1025+4: REM OFFSET
 45 X=FNB(I)+Y:POKEI+1,INT((I+X)/256):POKEI,I+X-256*INT((I+X)/256)
 50 I=I+X:IF FNA(I)<>0 AND I<E-2 THEN 45
 55 IF FNA(I)=0 AND I<E-2 THEN 35
 60 PRINT"LINK-COMPLETE":END
READY.
```

this point the end of BASIC is signaled. The next two bytes are pointers to the start of BASIC text for the second program as originally saved to disk. If the first line of the second program is moved forward in memory by four bytes, then the last line of the first program will point to the first line of the second program and the linking process will be nearly complete. In order to finish the linking procedure, the forward pointers of the second program must be recalculated to compensate for the relocation of code. This is done conveniently using the ROM chaining routine at $C442.

To use the attached machine code, first use the disk Copy command to concatenate up to four program pieces. For example, to append parts A, B and C to form program D, the following command is executed (after loading the DOS support):

►C0:D=1:A,1:B,1:C

Next load the machine code and the concatenated program D into memory and type:

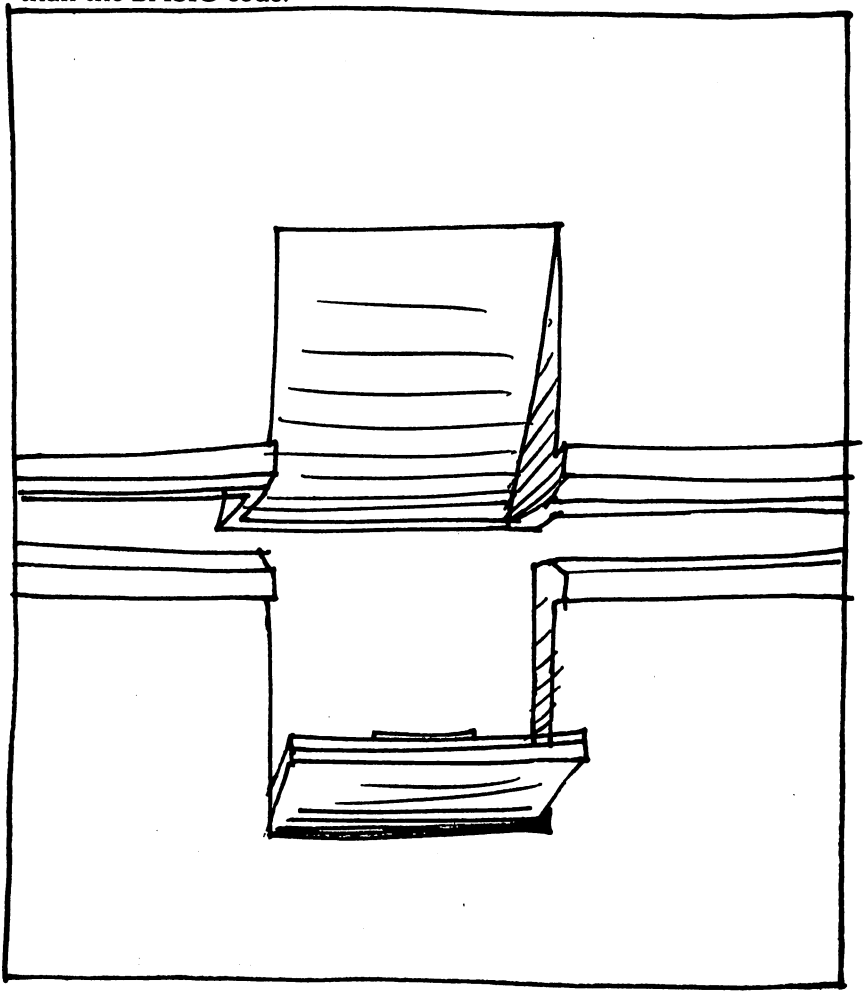
SYS826

The programs are linked and the message: "READY." appears.

For comparison, I have written a BASIC program to link concatenated programs. In this case the linking program must be the first program concatenated. Once concatenated, the new program is loaded into memory and then linked by typing:

RUN

The linking program is then deleted, leaving only the desired appended program. Comparison of these two procedures revealed that the machine language code ran nearly a thousand times faster than the BASIC code.



Using Direct Access Files With The Commodore 2040 Dual Drive Disk Part One

Chuck Stuart

How to use the random access capabilities of the 2040 disk drive.

One of the main advantages of using direct access files is the ability to access any record in a file directly without having to read through the entire file. With direct access, the last record in a file can be located and read into memory just as fast as the first record. Also, any record in a direct access file may be read into memory, updated, and then written back to the file without disturbing the other records in the file.

Although true direct access files are not directly supported in the current 2040 Disk Operating System, Commodore has provided a series of disk utility commands that will, in effect, allow direct access file processing. The difference is that, instead of the DOS keeping up with the track and sector addresses of each record in the file, a separate sequential file must be maintained to hold the record keys and address pointers. If for instance, the direct access file is a Customer account file keyed by account number, then the sequential file would hold an account number for each record in the account file plus the track and sector addresses for each record. This sequential file must be loaded into an array in memory before any processing of the direct access file can take place. To access a specific account, the array must be searched for the desired account number and then the corresponding track and sector numbers are used to directly access the record.

If the 2040 supported true direct access file processing, it would only be necessary to indicate the account number in the INPUT# or PRINT# statement and the DOS would keep up with the track and sector addresses in its own directory. Hopefully this will be implemented in a later version of the DOS.

It will probably be a little easier to understand and successfully use direct access files if you understand how a disk is laid out in

tracks and sectors. Each disk has 35 tracks, each track is divided into from 16 to 20 sectors, and each sector holds 256 bytes of data. Each byte will hold one character. Since an entire sector is read from or written to the disk at a time, sectors are generally referred to as data blocks or simply "blocks." Tracks and sectors do not physically exist on the disk, but are electronically impressed upon the surface material of the disk during the NEWing process, hence the expression "soft sectored." Track 18, being centrally located in the middle of the disk, is used by the 2040 DOS to hold the directory. The remaining 34 tracks are available to the user. If you're having trouble visualizing the tracks and sectors on a disk, think of the disk as a bull's eye target and the rings on the target as the tracks on the disk. Now if you cut the target into pie shaped wedges, you can see how the tracks are divided into sectors or data blocks.

Reading data into your program from the disk or writing data to the disk from your program using direct access is a two step process. To read data from a direct access file into your program, you must first load the data from the disk into one of the 256 byte disk buffers with the BLOCK-READ disk utility command. Once the data block has been successfully loaded into the buffer, it can then be read into memory with a standard input# statement. The process is just the reverse when writing data from your program to a direct access file. You first write the data to a buffer using a PRINT# statement, then the data must be loaded from the buffer onto the disk with the BLOCK-WRITE disk utility command. It is important to understand this process. The BLOCK-READ command loads an entire 256 byte sector from the disk into a buffer and makes it available to your program through a standard INPUT# statement. The BLOCK-WRITE command takes the contents of an entire 256 byte disk buffer and loads it onto a sector of the disk. It makes no difference if the record contained only one byte of data, it still occupies one entire 256 byte sector on the disk. Later I will explain how to place multiple records in a sector using the BUFFER-POINTER disk utility command.

One other area to cover is the BLOCK AVAILABILITY MAP (BAM). This is a reference map used by DOS to keep up with which blocks are being used and which blocks are available to use. To keep DOS from overwriting your direct access files with sequential files, you must flag those blocks on the BAM so DOS will know they are being used. As we will see later, this is done with the BLOCK-ALLOCATE disk utility command.

Now that the general concept of direct access files and the way they work on the Commodore 2040 Dual Drive Disk has been explained, the actual coding necessary to do the job will be examined line by line. Lines 500 to 680 would be part of the main program while lines 1000 to 1520 are subroutines which execute the various disk utility commands as required. The subroutines will be examined first, then the main program.

Lines 1000-1090

This subroutine is called after each disk utility or read/write command to check the error channel, channel 15, to see if a disk error has occurred. If an error has occurred, the error number and error message are displayed along with the track and sector address where the error occurred. If the error number is 00 then no error occurred and control returns to the main program.

Lines 1100-1190

This subroutine is used to allocate (or reserve) one sector on the disk through the use of the Block-Allocate disk utility command in line 1110. The sector is flagged on the BAM so DOS will not use it later for storage of sequential files. Looking at line 1110, D is the disk drive number, T is the track number, and S is the sector number. These values must be preset in the main program. After line 1110 requests the allocation, line 1120 reads the error channel to see if an error has occurred. If no error has occurred, control returns to the main program. If the error number is 65, this means that the requested block has already been allocated. But lo and behold, DOS has been kind enough to locate the track and sector numbers of the next available block and place them in ET$ and ES$. These values are placed in T and S and we again request allocation. Two important points must be remembered. DOS does not automatically allocate the next available block. It just tells you where it is. To allocate the block you must reset T and S to the values returned in ET$ and ES$ and then reissue the Block-Allocate command in line 1110. The other thing to remember is that, for a block to be successfully allocated, a direct access file must be open when the Block-Allocate command is given and that the block will not actually be reserved on the BAM until that file is closed. Allocating a block will not keep you from writing on it. It just keeps DOS from writing on it.

Lines 1200-1220

This subroutine is used to free a previously allocated block. The Block-Free command is the exact opposite of the Block-Allocate command. In line 1210, D is the disk drive number and T and S

hold the track and sector address of the block to be freed. After the command has been executed, line 1220 sends control to the error channel routine. If no error occurred, control returns to the main program. This routine is used to delete records from a direct access file by immediately releasing the block back to DOS. There is, therefore, no need for periodic system housekeeping to reclaim unused disk space. As with the Block-Allocate command, a direct access file must be open when the Block-Free command is given, and the block is not actually flagged as available until the file is closed.

Lines 1300-1320

This subroutine is used to make a block on the disk available for reading by your program. In the Block-Read utility command, line 1310, CH holds the channel number. D holds the disk drive number, and T and S hold the track and sector addresses of the block to be read. When the command is executed, a 256 byte data block is read from the disk and placed in one of the disk buffers. The data can then be read into memory with a standard INPUT# statement. After the block is read in from the disk, line 1320 sends control to the error check routine and, if no error has occurred, control returns to the main program.

Lines 1400-1420

This subroutine uses the Block-Write utility command to write the contents of a 256 byte buffer onto the disk. Again, CH holds the channel number, D holds the disk drive number, and T and S hold the track and sector addresses of the sector where the data is to be placed. Before this routine is executed, data should be placed in the buffer using the PRINT# statement. After execution, control passes to the error check routine and then back to the main program.

Lines 1500-1520

This routine uses the Buffer-Pointer utility command to set the buffer pointer to the byte in the buffer where reading or writing is to begin. Correct use of this routine will allow multiple records per sector, giving more efficient utilization of disk space. In line 1510, CH is the channel number and BP is the byte pointer. If BP is set to a value less than 1, it will be treated as though it were set to 1. If set to a value greater than 255, it will wrap around and begin at 1 again. Setting BP to 260 has the same effect as setting it to 5. After execution, line 1520 directs control through the error check routine and back to the main program.

Lines 500 to 590

These lines show the coding necessary to write records to a direct access file. They would be part of the main program.

Line 510 opens the command/error channel, channel 15, and assigns it to file number 15. Channel 15 must be opened and assigned to a file before any communication between computer and disk can take place.

Line 520 sets the channel variable to 3 and the disk drive variable to 1. The channel can be set to any unused channel between 3 and 15. The drive number is set to 1 for the left drive or 0 for the right drive.

Line 530 opens file number 1 and assigns it to channel CH, in this case, 3. The "#" tells DOS that this is a direct access file.

Line 540 is used to locate the next available sector and allocate it on the BAM. T is set to 1 and S is set to 0 because that is the address of the first sector on the disk. If that sector has been allocated, the next available sector is automatically located and allocated by the subroutine in lines 1200 to 1290.

Line 550 sets the buffer pointer to 1 so DOS will begin writing at the first byte in the buffer.

Line 560 writes the record data to the buffer beginning at the byte referenced by the buffer pointer.

Line 570 writes the buffer to the disk sector previously allocated in line 540. At this point, T and S must be saved along with whatever record key is being used so that this record can be found on the disk later.

Line 580 closes the direct access file opened in line 530.

Lines 600 to 680

This subroutine contains the coding necessary to read records from a direct access file. It would be part of the main program.

Line 610 opens file number 1 and assigns it to the preset channel in CH. The "#" tells DOS that this is a direct access file.

Line 620 loads a block of data from the disk and places it in the buffer assigned to channel CH. T and S must be set to the address of the sector where the desired record is located.

Line 630 sets the buffer pointer to begin reading at the first byte in the buffer.

Line 640 reads the record data from the buffer into the program.

Line 650 checks the status word.

Line 670 closes the direct access file.

This program will run as is. It will write the numbers 1 through 10 to the disk and then read them back in. If you add a line

to the program that will print T, S, and the array A$ on the screen, you can verify that the correct data was written to and then read from the disk and even see to which sector it was written. Notice that each time the program is run, a new sector is allocated and used. These sectors will become wasted space on the disk unless you free them with the Block-Free command. Add a GOSUB 1200 at line 665 and notice that now the program reuses the same sector each time. Why? What would happen if you moved the GOSUB 1200 to line 675? Why?

Now we will explain how to write more than one record to a sector. If you've followed everything up to this point, especially the section on the Buffer-Pointer command, then you have probably pretty well figured it out for yourself.

If each record in a direct access file occupies one entire sector of the disk, then each disk will only hold a maximum of about 670 records. If each record contained only a few bytes of data, this would be totally unacceptable waste of valuable disk space. In order to achieve maximum use of the available disk space, we must pack the maximum number of records to a sector.

In order to do this it is necessary to reduce the record size to the minimum number of bytes that will store the necessary data. Most DOS allow data to be written to the disk in binary format like the data is stored in memory. In other words, integer data requires two bytes of disk space and floating point data requires five bytes. Although 2040 DOS is an excellent first release version, this type of disk packing is one the standard DOS features not supported. Data is written to the disk in the same form it is written on the screen. Each character takes one byte of disk space. In addition, numeric data includes leading and trailing blanks. For this reason it is usually more efficient to write data to the disk in string format. String data occupies one byte of disk space for each character in the string. In addition, if the record contains more than one data field, then each field must be followed by a CARRIAGE RETURN, CHR$(13), field delimiter. This requires one extra byte per field. If each field in the record is always the same size, in other words the record contains no string fields such as CUSTOMER NAME that vary in size from record to record, then all the fields can be concatenated into a single string field before writing the record to the disk. This could result in a considerable saving since no field delimiters would be required. Upon reading the record back in, it could be split up into the original fields with the MID$ statement.

Once the maximum record size has been determined, divide

```
500 REM WRITE A DIRECT ACCESS RECORD
510 OPEN 15,8,15 :GOSUB 1000
520 CH=3 :D=1
530 OPEN 1,8,CH,"#" :GOSUB1000
540 T=1 :S=0 :GOSUB 1100
550 BP=1 :GOSUB1500
560 FOR I=1 TO 10 :PRINT#1,I CHR$(13); :NEXT I
570 GOSUB 1400
580 CLOSE 1
600 REM READ A DIRECT ACCESS RECORD
610 OPEN 1,8,CH,"#" :GOSUB 1000
620 GOSUB 1300
630 BP=1 :GOSUB 1500
640 FOR I=1 TO 10 :INPUT#1, A$(I)
650 IF ST THEN I=10
660 NEXT I
670 CLOSE 1
690 END
1000 REM ERROR CHANNEL INPUT ROUTINE
1010 INPUT#15, EN$,EM$,ET$,ES$
1020 IF EN$="00" GOTO 1090
1030 PRINT " DISK ERROR #" EN$ " " EM$ " " ET$ " " ES$
1040 INPUT " CONTINUE?  "; A$
1050 IF A$<>"Y" THEN STOP
1090 RETURN
1091 REM
1100 REM ALLOCATE 1 D/A BLOCK
1110 PRINT#15,"B-A";D;T;S
1120 INPUT#15,EN$,EM$,ET$,ES$
1130 IF EN$="00" GOTO 1190
1140 IF EN$="65" THEN T=VAL(ET$) : S=VAL(ES$):GOTO 1110
1150 GOTO 1030
1190 RETURN
1191 REM
1200 REM FREE 1 D/A BLOCK
1210 PRINT#15, "B-F";D;T;S
1220 GOTO 1000
1291 REM
1300 REM READ D/A BLOCK
1310 PRINT#15, "B-R";CH;D;T;S
1320 GOTO 1000
1391 REM
1400 REM WRITE D/A BLOCK
1410 PRINT#15, "B-W";CH;D;T;S
1420 GOTO 1000
1491 REM
1500 REM SET BUFFER POINTER
1510 PRINT#15, "B-P";CH;BP
1520 GOTO 1000
```

the record size in bytes into 255 to determine the maximum number of records that can be stored on a single sector of the disk. For

example, if each record in the file has been determined to have a maximum length of 20 bytes including all necessary field delimiters, then by dividing 20 into 255 we see that we can store 12 records per sector. Since the zero byte is used by DOS as an EOI pointer, the first record begins in byte 1, the second record in byte 21, the third record in byte 31, etc. Now you will have to add a fourth field to your sequential pointer file. Besides record key, track address, and sector address, you must identify each record's position in the block. Then, to locate a specific record in the file, you would search the record key array for the desired record, use the corresponding track and sector addresses to read in the indicated sector, and then set the buffer pointer to the value in the corresponding record position field. Now you are ready to read the desired record into your program with a standard PRINT# statement.

Before winding this up, there is one other important area that should be covered and that is the correct way to write data to the disk. The following lines show several ways data can be written.

```
100 PRINT#1, A$,B, C%
200 PRINT#1, A$;B; C%
300 PRINT#1, A$ CHR$(13) B CHR$(13)C% CHR$(13);
400 FOR I = 1 TO 10:PRINT#1, A$(I):NEXTI
500 FOR I = 1 TO 10:PRINT#1, A$(I),:NEXTI
600 FOR I = 1 TO 10:PRINT#1, A$(I)CHR$(13);:NEXTI
```

Line 100

WRONG! Commas have the same skipping effect on the disk as they do on the screen. This would result in very inefficient use of disk space.

Line 200

WRONG! Semicolons are non printing characters and will not work as field delimiters. Any attempt to read A$ would read B and C% as well.

Line 300

CORRECT. This method will write a CARRIAGE RETURN, CHR$(13), field delimiter between each field and the semicolon on the end keeps OS from adding a trailing LINE FEED character to the last field.

Line 400

WRONG! The OS will add CARRIAGE RETURN and LINE FEED characters to each field. The CARRIAGE RETURN character is desired but the LINE FEED will become the first character in the following field and can cause numerous problems.

Line 500

WRONG! Same reason as line 100.

Line 600

CORRECT. The required CARRIAGE RETURN character is inserted between each field in the record and the semicolon keeps the OS from adding a LINE FEED character. The PET Operating System treats all data the same no matter if it is printing to screen, disk, or printer. For this reason, the last field in every PRINT# command should be followed by a semicolon to keep the OS from adding a LINE FEED character to the output data string. This LINE FEED character will become the first character in the following field and will cause all kinds of headaches. It will crash your program with a data check error if you attempt to read the field in numeric format and can lead to erroneous comparisons if read in string format. This is true whether you are using direct access or sequential files. Data is much easier to read correctly from the disk if it was written correctly to the disk.

You should now be well versed in the theory of using direct access files on disk. Next comes the fun part, gaining actual experience reading and writing direct access files on your disk. Start with the program in lines 500-680 plus the subroutines in lines 1000 to 1520. When you are sure you know exactly what each line does, you can start experimenting around, adding lines, etc. When the program crashes, and it probably will several times, back up and don't try anything new until you know exactly what went wrong. Before you know it, you'll be the club expert on 2040 direct access files.

Part Two

We have attempted to explain in detail how to use and understand Direct Access Files with the new Commodore 2040 Dual Drive Floppy Disk. Using and understanding Direct Access file organization is mandatory if you plan to develop any serious business software for the Commodore microcomputer system.

We will now expand on the principles previously covered and also try to answer some of the questions we have received in the mail concerning those principles. References to line numbers in the following material refer back to the Direct Access coding explained in part one. It will help you to understand the following information if you refer back to that part.

Before getting started, it would probably be prudent to point out what seemed an error to many readers. Some thought that the GOTO 1000 in lines 1220, 1320, 1420, and 1520 should have been a GOSUB 1000 instead. On glancing over the coding this might appear to be true, but a more careful examination of the logic flow will show that each of the disk utility routines beginning at lines 1200, 1300, 1400, and 1500 flow into the error channel read routine beginning at line 1000 and the RETURN in line 1090 is the logical return path for each of these subroutines.

If you own a 2040, you should have by now received the final version of the instruction manual. If not, call (408) 727-1130 and ask for one. Although much more professional-looking in appearance, this final version offers little more useful information than the temporary version, especially in the area of Direct Access file organization. In light of this shortcoming, we will continue to pass along what practical information we uncover during the continued development and support of our business software systems. At the same time, we would appreciate receiving any additional information and user hints that others might discover.

Updating Direct Access Files

As we pointed out in part one, one of the best reasons for using Direct Access files is that it gives you the ability to read in any record in a file, update the information contained in that record, and then write that record back to the file without disturbing any other records in the file. The records can be accessed in any other records in the file. The records can be accessed in any order regardless of their physical order on the disk. The last record in the file can be found and read as easily and as fast as the first record in the file.

Unfortunately, the BLOCK-READ and BLOCK-WRITE disk utility commands that we previously covered do not lend themselves well to this type of file updating. Early on in our software development we discovered this problem. The B-W command places the value of the current buffer pointer in the zero byte of the block and the B-R command uses this value to set the STATUS WORD to 64. During this process the record data is affected if an attempt is made to update more than one record per OPEN statement. The only answer was to OPEN and then CLOSE the file each time a record in the file was to be updated. Needless to say, this method greatly slowed processing — not because of the extra time required to OPEN and CLOSE the files, but because of the time required for the disk READ/WRITE HEAD to physically move to the disk Directory Track, track 18, to update the BLOCK AVAILABILITY MAP (BAM) and then return to the processing track.

Fortunately there is a better method. The USER command U1 is used rather than the BLOCK-WRITE command. The following two lines should replace the corresponding lines of BASIC coding shown in part one.

```
1310 PRINT#15,    U1 ;CH;D;T;S
1410 PRINT#15,    U2 ;CH;D;T;S
```

The USER command, U1, performs the same function as the BLOCK-READ command, B-R, except that a 255 character block is assumed since any character count stored in the zero byte is ignored. The USER command, U2, is identical to the BLOCK-WRITE command, B-W, except that the contents of the zero byte are unchanged when the block is written to the disk. When using the U1 and U2 commands for reading and writing Direct Access disk files, you must always set the buffer pointer to the desired position before issuing a PRINT# or INPUT# command. This is accomplished through the use of the BUFFER-POINTER command, B-P, as explained in part one. With a little practice you should now be able to update Direct Access files on the 2040 with no problems.

Detecting A Disk Full Condition Using Direct Access Files

In explaining how to create Direct Access files, we left out one rather important detail. That was how to tell when the disk is full. Go back to lines 1100-1190 and study the associated description of how the BLOCK-ALLOCATE command works. Remember that this command is used for finding and allocating the next available disk sector in creating and expanding a Direct Access file. The track and sector address of the next available block is returned in the

third and fourth parameters of the Command Channel. This is ET$ and ES$ in line 1120 of our example. The block located at that track and sector is then allocated and becomes part of our Direct Access file.

All this is great, but what happens when all of the available disk sectors have been allocated? How does DOS manage to inform us of this rather important event? Simple. When no more disk sectors are available then DOS returns 00's in ET$ and ES$ instead of the address of the next available sector. Be sure to check EN$ for a 65 to ensure that some type of disk error has not occurred that might also cause ET$ and ES$ to be set to 00. The following lines of BASIC coding should be inserted in the program example in part one.

```
1140 IF EN$<>65  GOTO 930
1150 IF ET$=00 AND ES$=00
GOTO 9000
1160 T= VAL(ET$): S = VAL(ES$):
GOTO 1110
```

Notice that the original line 1140 is replaced by the new one. Line 1140 now checks for the correct error number to ensure that some unexpected disk error has not occurred. If one has occurred, if EN$ is any value other than 65, then control is passed to the error handling routine. Notice that the branch is to line 930 rather than the usual line 900. This is because the error channel has already been read in line 1120. Line 1150 checks to see if the disk is full and, if so, control is passed to the appropriate routine at line 9000. Line 1160 resets T and S to the track and sector address of the next available sector and branches back to request allocation.

```
100 REM THIS IS AN EXAMPLE OF DIRECT
105 REM ACCESS FILE UPDATING.
106 REM
110 REM LINES 200 TO 299 READ A RECORD
115 REM FROM A DIRECT ACCESS FILE,
120 REM UPDATE THE RECORD, THEN WRITE
125 REM THE RECORD BACK TO THE FILE.
126 REM
130 REM LINES 1000 THROUGH 1520 WERE
135 REM DISCUSSED IN DETAIL IN PART
140 REM ONE OF THIS SERIES.
150 REM
200 REM UPDATE A DIRECT ACCESS RECORD
210 CH=4: D=0: BP=1: CR$=CHR$(13)
220 OPEN4,8,4,"#": GOSUB 1000
230 T=15: S=12: GOSUB 1300: GOSUB 1500
```

```
240 INPUT#4,A$,B$,C$,D$,K
250 A$="TEST": K=K+1
260 GOSUB 1500
270 PRINT#4,A$ CR$ B$ CR$ C$ CR$
275 PRINT#4,D$ CR$ K CR$
280 GOSUB 1400
290 CLOSE 4
299 END
1000 REM ERROR CHANNEL INPUT ROUTINE
1010 INPUT#15, EN$,EM$,ET$,ES$
1020 IF EN$="00" GOTO 1090
1030 PRINT "DISK ERROR #" EN$ " " EM$;
1031 PRINT " " ET$ " " ES$
1040 INPUT "CONTINUE? "; A$
1050 IF A$<>"Y" THEN STOP
1090 RETURN
1091 REM
1300 REM READ A D/A BLOCK
1310 PRINT#15, "U1";CH;D;T;S
1320 GOTO 1000
1391 REM
1400 REM WRITE A D/A BLOCK
1410 PRINT#15, "U2";CH;D;T;S
1420 GOTO 1000
1491 REM
1500 REM SET BUFFER POINTER
1510 PRINT#15, "B-P";CH;BP
1520 GOTO 1000
```

File Conversions On The Commodore 2040 Drive

Hal Wadleigh

This article complements Robert W. Baker's "WordPro Converter." Now you can process a WordPro "File Cabinet" or provide mailing label merging.

The vast majority of business systems using PETs and CBMs are attached to a 2040 disk drive. One of the reasons is that Wordpro (the word processing system) can turn the CBM-2040 combination into one of the most cost-effective word processing machines on the market. The other applications which can also be programmed for the same equipment are a nice bonus, but Wordpro usually pays for the system.

The initial concept behind Wordpro was for the word processing function to be distinct from other functions. The software design was predicted on the assumption that Wordpro files would not have to be accessed by other programs. Consequently, the files were designed in program blocks — not as ASCII data files. Time has proven that assumption to be inaccurate. Many text files created in Wordpro contain information that is often necessary for other computer functions and Wordpro documents often need data from ASCII files created by other functions.

The problem has a fairly simple solution — two conversion programs. One translates Wordpro files into ASCII files and the other performs the reverse.

The attached program listings will not be very meaningful without some explanation of the way Wordpro files are structured. Wordpro files are programs, not sequential files. A carriage return by the operator puts a back-arrow into the text and moves the cursor to the start of the next screen line. Since the file is actually a program, the first two bytes of the file are a reverse-format integer designating the address at which the program should begin loading. Luckily, the 2040 DOS allows single character GETs from a program file. It may be OPENed in the same way a sequential file is opened, and the status word (dedicated variable ST) can be used to find the end of the file.

In the attached listing of the Wordpro to ASCII conversion program, the subroutine at line 10000 sets up the character

conversion table. Characters in Wordpro files are in screen code, not ASCII. The subroutines at 1000 is the error checking and end-of-file scanning routine. These two techniques are the key elements in file conversion for these kinds of files. The structural differences between the file types is handled by simply throwing out the two pointer bytes at the beginning of the file. The program listed does this with a pair of initial GETs and is processed by replacing it with a carriage return and throwing out the additional blank spaces behind the back arrow. That is the reason that it is necessary to keep track of the screen column from which the Wordpro character came (C% in the program).

The program which does the reverse conversion uses the same basic techniques, but re-inserts the pointer and padding blanks that are discarded in the other program.

Word processing is the foremost application for all microcomputers today. Now it doesn't have to be an entirely separated function on 2040 systems, but can be integrated into a cohesive data processing system for small businesses. Invoices generated in Wordpro can be converted and merged into an order-processing system. Block file lists generated and maintained in Wordpro can be converted for a demographic analysis. Mailing list label data can be converted into Wordpro block files for use in customized mass mailings.

There are other instances where the same type of file conversion techniques may be useful — converting the CBM altered ASCII set to standard ASCII for output to a printer is one example. The principle involved is to do as much of the conversion as possible in a table and to program only those functions that cannot be mapped. It's easier and faster that way.

```
5 POKE59468,14:GOSUB10000
7 PRINT"ĥWORDPRO TO ASCII CONVERSION":
    ¬PRINT
10 INPUT"WORDPRO FILE NAME  .<<<";F$:
    ¬IFF$="."GOTO10
20 INPUT"DRIVE NUMBER  .<<<";A$:A=ASC(A$)
    ¬-48:IFA<0ORA>1GOTO20
30 INPUT"ASCII FILE NAME  .<<<";AF$:
    ¬IFAF$="."GOTO30
40 INPUT"DRIVE NUMBER  .<<<";A$:B=ASC(A$)
    ¬-48:IFB<0ORB>1GOTO40
50 A$=CHR$(A+48):B$=CHR$(B+48):OPEN15,8,
    ¬15:PRINT#15,"I"+A$
60 IFB$<>A$THENPRINT#15,"I"+B$
70 F$=A$+":"+F$+",P,R":REM DRIVE#:
```

```
      ¬FILE NAME,PROGRAM,READ ACCESS
80 OPEN1,8,2,F$:GOSUB1000:REM OPEN AND ¬
      ¬CHECK
90 AF$=B$+":"+AF$+",S,W":REM DRIVE#:
      ¬FILE NEME,SEQUENTIAL,WRITE ACCESS
100 OPEN2,8,3,AF$:GOSUB1000
110 GET#1,A$:GOSUB1000:REM SKIP POINTER,
      ¬ LOW BYTE
120 GET#1,A$:GOSUB1000:REM SKIP POINTER,
      ¬ HIGH BYTE
130 C%=39:REM INITIALIZE COLUMN COUNTER
140 GET#1,A$:C%=C%+1:IFC%>39THENC%=0
150 IFA$=CHR$(31)GOTO170:REM CARRIAGE ¬
      ¬RETURN
160 PRINTT$(ASC(A$));:PRINT#2,T$(ASC(A$))
      ¬;:GOSUB1000:GOTO140
170 PRINT:PRINT#2,CHR$(13);:IFC%=39GOTO14
      ¬0:REM NO PADDING
180 FORI=C%TO38:GET#1,A$:GOSUB1000:
      ¬IF(ASC(A$)AND63)<>32THENPRINT:
      ¬PRINT"rFILLER ERROR":STOP
190 NEXT:GOTO130:REM FILLER(CHARACTER 32 ¬
      ¬OF 160) HAS BEEN DELETED
200 END
1000 SA%=ST:REM TEMPORARY STORAGE OF ¬
      ¬STATUS WORD
1010 INPUT#15,E,E$,T,S:IFE<1GOTO1040:
      ¬REM DISK ACTION OK
1020 PRINT:PRINT"rDISK ERROR";E;E$:
      ¬PRINT"TRACK=";T;"SECTOR=";S:
      ¬PRINT"STATUS=";SA%
1030 CLOSE1:CLOSE2:CLOSE15:END
1040 IFSA%=0THENRETURN:REM ALL OK
1050 M$="STATUS ERROR="+STR$(SA%):
      ¬IFSA%=64THENM$="FILE CONVERTED"
1060 PRINT:PRINT"r";M$:GOTO1030
1070 REM END OF ERROR CHECKING SUBROUTINE
10000 DIMT$(255):REM TRANSLATION ARRAY
10010 FORI=0TO31:T$(I)=CHR$(I+64):NEXT
10020 FORI=32TO63:T$(I)=CHR$(I):NEXT
10030 FORI=64TO95:T$(I)=CHR$(I+128):NEXT
10040 FORI=96TO127:T$(I)=CHR$(I+64):NEXT
10050 FORI=128TO255:T$(I)="r"+T$(I-128)+"
      ¬î":NEXT
10060 RETURN
```

COMPUTE NOTES: The pointer found at the beginning of the WORDPRO II files will vary depending on the number of lines of main text you specified. Since WORDPRO II doesn't use this pointer when loading files, you could increase the workspace in the ASCII TO WP program by reducing the value assigned to WP in line 20. Just make sure that you declare enough main text in WORDPRO to

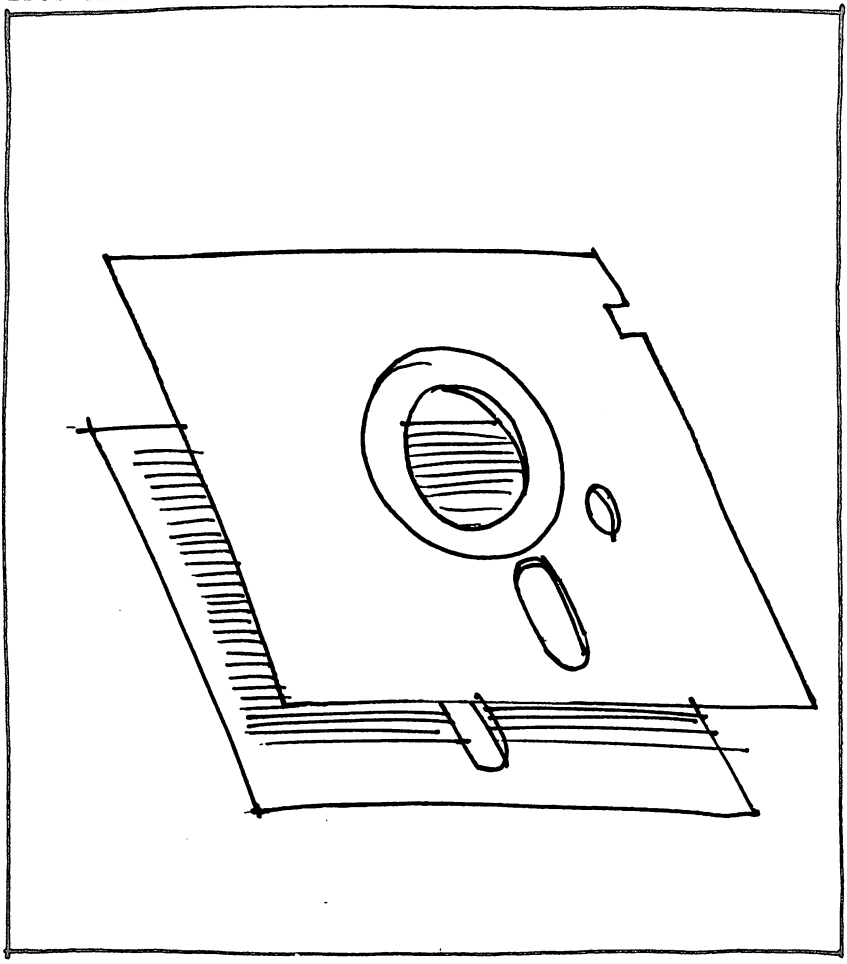
accommodate the file.

Use abbreviations in lines 180 and 1020 of the WORDPRO TO ASCII program to keep from exceeding the limit of 80 characters per line, i.e. use "gE" for "get#" and "?" for "print".

Also, the "~" character in line 95 of the ASCII TO WORDPRO program represents a backarrow character.

```
5 POKE59468,14:GOSUB2000
7 PRINT"ñASCII TO WORDPRO CONVERSION":
      ¬PRINT
10 EL=PEEK(52):EH=PEEK(53):REM POINTERS ¬
      ¬TO END OF MEMORY
20 WP=12296:EW=EL+256*EH-1:REM $3008--BEGI
      ¬NNING & END OF WORDPRO WORKSPACE
30 INPUT"ASCII INPUT FILE  .<<<";F$:
      ¬IFF$="."GOTO30
40 INPUT"DRIVE  .<<<";D$:D=ASC(D$)-48:
      ¬IFD<0ORD>1GOTO40
50 INPUT"WORDPRO FILE NAME  .<<<";WF$:
      ¬IFWF$="."GOTO50
60 INPUT"DRIVE  .<<<";D$:WD=ASC(D$)-48:
      ¬IFWD<0ORWD>1GOTO60
70 F$=RIGHT$(STR$(D),1)+":"+F$+",S,R":
      ¬WF$=RIGHT$(STR$(WD),1)+":"+WF$+",P,
      ¬W"
80 OPEN15,8,15:PRINT#15,"I":GOSUB1000:
      ¬OPEN1,8,2,F$:GOSUB1000
82 OPEN2,8,3,WF$:GOSUB1000:L%=0:CB=WP+2:
      ¬R%=0
85 PRINT#2,CHR$(8)+CHR$(48);:GOSUB1000:
      ¬REM INITIAL POINTERS
90 GET#1,A$:SA=ST:GOSUB1000:IFA$=""GOTO170
95 IFA$=CHR$(13)THENPRINT"~":GOTO120
96 PRINTA$;
100 IFA$="r"THENR%=1:GOTO170
110 IFA$="f"THENR%=0:GOTO170
120 BV=T%(ASC(A$)):IFBV=0ANDA$<>"@"GOTO170
125 IFR%=1THENBV=BV+128
130 PRINT#2,CHR$(BV);:CB=CB+1:L%=L%+1:
      ¬IFL%>39THENL%=0
140 IFA$<>CHR$(13)GOTO160
150 IFL%>0THENFORI=L%TO39:PRINT#2," ";:
      ¬CB=CB+1:NEXT:L%=0:R%=0
160 IFCB>EWGOTO210:REM END OF WORKSPACE
170 IFSA<1GOTO90
180 IFSA=64GOTO200
190 PRINT:PRINT"rSTATUS ERROR";SA:END
200 PRINT:PRINT"rFILE CONVERTED":CLOSE2:
      ¬GOSUB1000:CLOSE1:GOSUB1000:CLOSE15:
      ¬END
210 PRINT:PRINT"rWORKSPACE FULL--CHAINING ¬
```

```
      ¬FILE":CLR2:GOSUB1000
220 WF$=LEFT$(WF$,LEN(WF$)-4)+STR$(C%)+",
      ¬P,W":GOTO82
1000 INPUT#15,E,E$,T,S:IFE<1GOTO1020
1010 PRINT:PRINT"rDISK ERROR";E;E$:
      ¬PRINTT,S,SA:END
1020 IFSA<1ORSA=64THENRETURN
1030 PRINT:PRINT"rSTATUS ERROR";SA:END
2000 DIMT%(255)
2010 T%(13)=31:REM WORDPRO RETURN MARKER
2020 FORI=64TO95:T%(I)=I-64:NEXT
2030 FORI=32TO63:T%(I)=I:NEXT
2040 FORI=192TO223:T%(I)=I-128:NEXT
2050 FORI=224TO255:T%(I)=I-64:NEXT
2060 RETURN
```



Using Disk Overlays On The PET

Marty Franz

Overlaying or "chaining" lets you write and use programs that are much larger than your available memory.

If you've written very large BASIC programs on the PET, there've probably been times when you've wanted to break them into smaller pieces to save storage or make maintenance easier. As an example of this, consider disk utilities: you don't need to have the code necessary to copy files and check for errors loaded into storage along with the code to list the disk directory, since you will only be performing one of those functions at a given time. Instead, it'd be nice to write small "copy files" and "list directory" programs and just load the one needed. To make accessing these programs more convenient, a master program could then decide which one to load, based on your input. When the little utility programs were done, they'd reload the master program again to allow another selection.

This approach to program design is called "overlay segmenting." It has several advantages over putting every desired function into a single large program. One is storage efficiency: only the code needed to perform a specific function is loaded in storage at any given time. Another is modularity: when a bug is found you only have to modify a single, small program. Also, if you want to add another function you only have to write the program that implements it and make the master program "aware" of how to load it.

Fortunately, the PET allows one program to load another by letting you put LOAD statements in BASIC programs. It will even suppress the normal SEARCHING FOR and LOADING messages, so the user won't notice that another program is being loaded.

Unfortunately, there's a serious restriction on this: the overlay program must be smaller than the program loading it! This is because the pointers that tell BASIC how large the program is and where the variables are kept aren't reset when a LOAD is done from another program. The Microsoft people have done this so that the new program can access the previous one's variables. But it means that the master program has to be the largest program in our utility package. What if we want to add an "edit files" function? It'd

be nice to use all the memory available to us for keeping text while we were editing it. With the master program hogging storage to insure that its overlays will always be smaller, this will severely limit what's available to our editor.

To have a really useful disk overlay scheme, then, we have to get around this restriction on program size. For that, we have to know how BASIC programs are stored. Addresses 42 and 43 on new PETs are a pointer (low and high bytes) to where variable storage starts in a BASIC program. It also tells where the program text ends. If we poke these addresses with a larger pointer value, we've effectively increased the size of our program.

So, a way to insure that our program is always bigger than the one we're going to load next is to set the "program end" pointer right before the LOAD statement. What's the largest possible value we can give to its pointer? It's the highest RAM memory address, kept for us at address 52 and 53. We want to make it one page (256 bytes) less than that, actually, because BASIC might need interim storage for variables.

When the new program is loaded and begins running, the first thing it will have to do is tell BASIC how big it really is, so that the rest of storage can be freed up for use by its variables. Luckily, the actual length of a program is kept by the LOAD routines at addresses 201 and 202. So, we reset the pointer at address 42 to this value before we do any processing in our overlay.

The only flaw in this scheme is the previous program's variables. BASIC keeps track of where variable storage begins and ends, and it's based on the end-of-text pointer. Each time we mess with this pointer we have to get BASIC to clean up variable storage for us. The CLR statement will do that.

Our overlay scheme is now complete. Whenever we load an overlay, we make our program as large as possible beforehand to avoid the length restriction:

```
996 POKE 42,PEEK(52)
997 POKE 43,PEEK(53)-1
998 CLR
999 LOAD "next",8
```

And, when the overlay begins execution, we first reset the program-end pointer to the actual program size:

```
1 POKE 42,PEEK(201)
2 POKE 43,PEEK(202)
3 CLR
```

The CLR statements clean up the variables for us, and we allow one

page less than the largest RAM address for interim storage of BASIC variables.

There are two "gotchas" involved in this overlay scheme, however. The first is that if we want to pass variables between overlays we're out of luck, because the CLRs will clobber whatever was kept in storage. In a future article, I'll discuss a secure way to pass parameters between programs. For now, we can POKE variables into a protected storage area like the second cassette buffer and have them remain intact during the LOAD process. We'll have to do this for the name of the program you're loading if it's going to be kept in a string. That's why we kept a page of storage free for variables when we increased the program's size. After the pointer is reset, we need to fish the name of the program out of protected storage as a string so we can do the LOAD. The sample master program shows how this is done.

The second "gotcha" relates to writing and testing overlays. It's probably a good idea to omit the entry and exit linkage from them while they're being tested. Why? Each time the program is run, it'll reset its end-of-program pointer to the value it had when it was last loaded. This is very bad when you make a lot of changes and the program size is altered dramatically. You could lose all your changes (at best) or mess up your program to the point where you can't do anything with it (at worst). When debugging an overlay, type NEW before loading it, and be sure to save a copy before running it again.

There are many applications for overlays. For example, the master program could use light pen input to select the overlays, simplifying the user's interface to the PET. If the overlays were carefully written to take all their parameters from memory, the master program could read input from a file, and complex functions could be built up from lists of simpler ones. A file could tell the master program to first assemble, then run, a 6502 machine language program by calling assembler and loader overlays. This is called "batch" or "command language" processing and is done on much larger computer systems. Suffice it to say that with the ability to do overlay segmenting, the power of the PET for serious programming is greatly enhanced.

```
100 REM  SAMPLE MASTER PROGRAM
110 REM  USING PET DISK OVERLAYS
120 REM
130 POKE 42,PEEK(201):POKE 43,PEEK(202)
140 CLR
150 REM  INITIALIZE PROGRAM
```

```
160 DF$="-":SI$="  "+DF$+"■■■"
170 OPEN 15,8,15
180 REM  ASK FOR PROGRAM NAME
190 PRINT:PRINT:PRINT"PROGRAM";SI$;
200 INPUT P$
210 :IF P$=DF$ THEN 190
220 :IF P$="BYE" THEN NEW:END
230 REM  CHECK DISK FOR PROGRAM
240 OPEN 1,8,2,P$+",P,R"
250 GOSUB 500
260 CLOSE 1
270 IF E=0 THEN 300
280 :PRINT:PRINTEM$
290 :GOTO 180
300 GOSUB 360
310 POKE 42,PEEK(52):POKE 43,PEEK(53)-1
320 CLR
330 GOSUB 430
340 LOAD P$,8
350 END
360 REM  SAVE P$ IN TAPE BUFFER
370 POKE 827,LEN(P$):A=828
380 FOR I=1 TO LEN(P$)
390 :POKE A,ASC(MID$(P$,I,1))
400 :A=A+1
410 NEXT I
420 RETURN
430 REM  GET P$ FROM TAPE BUFFER
440 P$="":A=828
450 FOR I=1 TO PEEK(827)
460 :P$=P$+CHR$(PEEK(A))
470 :A=A+1
480 NEXT I
490 RETURN
500 REM  CHECK DISK ERRORS
510 E=0
520 INPUT#15,EN,EM$,ET,ES
530 IF EN=0 THEN RETURN
540 E=1
550 RETURN
```

Variable-Field-Length Random Access Files On The 2040 Disk Drive

Peter Spencer

A sophisticated technique to make the most of your 2040's random access capabilities.

Do you have voluminous file storage needs, but hate to see a large fraction of each disk eaten up by the empty space that seems to be an inherent feature of most random access programs?

This program shows how to write variable field length random access files on the 2040 disk drive. The density of packing is truly amazing. Compare it to the density achieved by any fixed length program you have, including the lengthy relative record program in the 2040 User's Manual.

The writing to disk is done in lines 41 to 77, and the retrieval from disk is in lines 82 to 106. The rest of the program is a driver routine patched on from a longer program of mine.

For this sample program, I have used the line number as the key for each field. You can easily use some other key, and have more than one field per key. In that case, you must change the output to the key file (lines 71-77) so that it contains the number of keys used, each key, the number of fields for that key (if variable), and the track, sector, and buffer pointers for each field within that key. Lines 88-95 would have to be similarly changed.

Yes, you read the above correctly, you can even have a variable number of fields per key! Such a variable field number, variable field length program can be of considerable use if you want to store abstracts, test questions, criterion-referenced test questions (using the criterion or instructional objective code "number" as the key), or parts inventory (you could use the machine name as the key, and each part as a field, with subfields for cost, price, or onhand, backordered, and so forth).

The driver routine I have used can be considerably shorter if you wish to use regular input rather than the bullet-proof and hyphenation-proof form provided in lines 118-133. There, a line-overrun on input from the keyboard (detected in line 125) results in the entire word being removed to the next consecutive line

(accomplished in lines 128-133 and 119).

```
1 CLR
2 PRINT"ĥ↓VARIABLE FIELD LENGTH FILES ON ¬
     ¬THE 2040↑          ↓PETER SPENCER"
3 GOSUB1Ø8:MK=Ø:LL=8Ø
4 DIMPA(3ØØ):DIMTA(3ØØ):DIMSA(3ØØ)
5 NLS=1:D=Ø:F=Ø:X=Ø:Y=Ø:T=Ø
6 SP$="                                    ¬
     ¬    "
7 M$=CHR$(13)
8 S$="":Z$="":IN$="":DIMA$(3ØØ):OPEN15,8,
     ¬15
9 REM:  PROGRAM ENTRY
1Ø PRINT"ĥ↓S↑TART NEW FILE, OR ↓W↑ORK ON ¬
     ¬OLD FILE? ";
11 GOSUB33
12 PRINT"NAME OF FILE ";:GOSUB119:
     ¬A$(1)=IN$
13 IFS$="S"THENGOTO22
14 GOTO83
15 REM:  SHOW FILE ENTRIES
16 FORK=1TONLSSTEP15:F=K:D=K+14
17 FORI=FTOD:PRINTI;TAB(6);A$(I):NEXTI
18 PRINT"ĥ";SP$;SP$;SP$
19 PRINT"ĥ↓S↑CROLL NEXT 15 LINES, OR ¬
     ¬↓E↑XIT? ";:GOSUB33:IFS$="E"THENK=NLS
2Ø PRINT"ĥ↓↓↓";:NEXTK
21 REM:  SHOW MENU
22 PRINT"ĥ";SP$;SP$;SP$
23 PRINT"ĥ↓R↑EAD IN, ↓O↑UTPUT, ↓T↑YPE, ";
24 PRINT"↓S↑CROLL, ";
25 PRINT"↓E↑XIT?";:GOSUB33
26 IFS$="E"THEN79
27 IFS$="T"GOTO11Ø
28 IFS$="O"GOTO42
29 IFS$="R"GOTO83
3Ø IFS$="S"THENPRINT"ĥ↓↓↓";:GOTO16
31 GOTO22
32 REM:  GET UTILITY
33 GETS$:IFS$=""THEN33
34 PRINTS$:RETURN
35 REM:  READ ERROR CHANNEL
36 INPUT#15,EN$,EM$,ET$,ES$
37 IFEN$="ØØ"THENRETURN
38 PRINT"ERROR ON DISK"
39 PRINTEM$;EN$,ET$,ES$
4Ø CLOSE6:CLOSE7:CLOSE15:END
41 REM:  OUTPUT ROUTINE
42 IFMK<>ØTHEN46
43 PRINT"INSERT DISK IN LEFT DRIVE & TYPE ¬
     ¬GO";:GOSUB33
```

```
44 PRINT#15,"I1"
45 OPEN6,8,6,"#":GOSUB35
46 PRINT"THERE ARE";NLS;"ENTRIES":MK=1
47 PRINT"STORE FROM ";:GOSUB119:X=VAL(IN$)
     ¬:PRINT"TO ";
48 GOSUB119:Y=VAL(IN$)
49 I=X
50 REM:  ALLOCATE 1 BLOCK
51 T=1:S=0
52 PRINT#15,"B-A";1;T;S
53 INPUT#15,EN$,EM$,ET$,ES$
54 IFEN$="00"THEN57
55 IFEN$="65"THENT=VAL(ET$):S=VAL(ES$):
     ¬GOTO52
56 GOTO38
57 BP=1
58 PRINT#15,"B-P:"6;BP:GOSUB35
59 PRINT#6,A$(I);M$;:GOSUB35:PRINTI;A$(I);
     ¬T;S;BP
60 PA(I)=BP:TA(I)=T:SA(I)=S
61 BP=BP+LEN(A$(I))+1
62 IF(LEN(A$(I+1))+1+BP)>255THEN67
63 I=I+1
64 IFI<=YTHEN58
65 PRINT#15,"U2:"6;1;T;S:GOSUB35
66 CLOSE6:GOTO72
67 PRINT#15,"U2:"6;1;T;S:GOSUB35
68 I=I+1
69 IFI<=YTHEN50
70 CLOSE6
71 REM:  OUTPUT KEY FILE, OVERWRITING OLD ¬
     ¬KEY FILE IF NECESSARY
72 OPEN7,8,7,"@1:"+LEFT$(A$(1)+SP$,
     ¬10)+".KEY01,S,W":GOSUB35
73 PRINT#7,NLS;M$;:GOSUB35
74 FORI=1TONLS
75 PRINT#7,TA(I);M$;SA(I);M$;PA(I);M$;:
     ¬GOSUB35
76 NEXTI
77 CLOSE7:GOSUB35
78 REM:  EXIT PROGRAM
79 PRINT"SHUT DOWN?";:GOSUB33
80 IFS$="N"GOTO22
81 CLOSE6:CLOSE7:CLOSE15:END
82 REM:
83 PRINT"rREAD KEYS AND FILE FROM DISK"
84 IFMK<>0THEN87
85 PRINT"INSERT DISK IN LEFT DRIVE & TYPE ¬
     ¬GO";:GOSUB33
86 PRINT#15,"I1":MK=1
87 OPEN7,8,7,"1:"+LEFT$(A$(1)+SP$,
     ¬10)+".KEY01,S,R":GOSUB35
```

```
88 INPUT#7,NLS:RS=ST:GOSUB35
89 PRINT"↓NLS=";NLS
90 PRINT" #  TR SE BP"
91 FORI=1TONLS
92 INPUT#7,TA(I),SA(I),PA(I):RS=ST:GOSUB35
93 PRINTI;TA(I);SA(I);PA(I)
94 NEXTI
95 CLOSE7:GOSUB108
96 REM:  READ FILE
97 OPEN6,8,6,"#":GOSUB35
98 FORI=1TONLS
99 PRINT#15,"U1:"6;1;TA(I);SA(I):GOSUB35
100 PRINT#15,"B-P:"6;PA(I)
101 GOSUB35
102 INPUT#6,A$(I):GOSUB35
103 IFTA(I)=0THEN106
104 PRINTI;A$(I)
105 NEXTI
106 CLOSE6:GOSUB108
107 GOTO22
108 FORI=1TO1000:NEXTI:RETURN:REM:
      ¬  DELAY LOOP
109 REM:  TYPE ROUTINE
110 PRINT"LENGTH OF LINE (MAXIMUM=80)";:
      ¬Z9$="80":GOSUB119:LL=VAL(IN$)
111 PRINT"ĥ↓TYPE NEW LINES";CHR$(13);"(TYP
      ¬E 'STOP' TO STOP))":PRINTSP$
112 D=NLS:IFD>=5THENF=D-4:GOSUB135:GOTO114
113 F=1:GOSUB135
114 PRINTNLS+1;CHR$(13);"↑";TAB(4)
115 GOSUB119:IFIN$="STOP"THEN22
116 A$(NLS+1)=IN$
117 NLS=NLS+1:GOTO111
118 REM:  BULLET-PROOF INPUT
119 IN$="":IFZ9$<>""THENPRINT"? ";Z9$;:
      ¬POKE167,0:IN$=Z9$:Z9$="":GOTO121
120 PRINT"? ";:POKE167,0
121 GETZ$:IFZ$=""THEN121
122 IFZ$=CHR$(13)THENPRINT" ":POKE167,1:
      ¬RETURN
123 IFZ$=CHR$(20)THENONSGN(LEN(IN$))+1GOTO
      ¬121,127
124 PRINTZ$;:IN$=IN$+Z$
125 IFLEN(IN$)>LLTHENGOSUB128:PRINT" ":
      ¬POKE167,1:RETURN
126 GOTO121
127 PRINTZ$;:IN$=MID$(IN$,1,LEN(IN$)-1):
      ¬GOTO121
128 FORZ9=LEN(IN$)TO1STEP-1
129 IFMID$(IN$,Z9,1)<>" "THEN133
130 Z9$=RIGHT$(IN$,LEN(IN$)-Z9)
131 IN$=LEFT$(IN$,Z9-1)
```

```
132 Z9=1
133 NEXTZ9:RETURN
134 REM:  SCREEN DISPLAY
135 FORI=FTOD:PRINTI;TAB(6);A$(I):NEXTI:
      ¬RETURN
READY.
```

PET/CBM Front Panel

Boyd Ray

Here is a novelty program for all PET/CBM owners. It displays, in realtime, the binary value of the data in sixteen user-selected memory locations. The display is placed on the upper part of the screen and is updated sixty times a second giving a front panel effect. The cursor is excluded from this field.

Sixteen arbitrary memory locations have been selected for inspection. You may change them as often as you wish by revising the data on the first three data lines and re-running the program. Prefix four-character hexadecimal locations with a "$."

Program 1 will run on all PET/CBM machines including the 80XX. Take extra care when typing the program and be sure to save it before attempting to run it. The machine code is self-modifying and is located just below screen memory.

When you have checked the listing a couple of times and you are certain it is right, run the program. If it works and your "blinking lights" front panel is blinking as it should be, jot down the SYS vectors for later use. Now press a few keys and observe the changes in the data or show junior how the clock counts in binary. You might want to refer to your favorite memory map from here on. Now try these direct commands:

```
POKE 59467,16:POKE 59466,15
FOR I=0TO255:POKE 59464,I:FOR J=1to25:NEXT J,I
```

Watch the shift register in action. Try different values in location 59466 and observe the change in wave shape.

When you've had your fill of this sort of thing, change the values on the first three data lines to 826 through 841 and add line 110;

```
110 POKE 826+16*RND(1),256*RND(1):GOTO110
```

Now re-run the program. This causes a "do nothing" random bit display which is similar to one of the props seen on the submarine "Seaview" in the old TV series.

If you prefer 1's and 0's in your display, change the 81 and 87 on data line 133 to 49 and 48 before saving the program. If the program is running, POKE (the disable vector + 133) with 49 and POKE (the disable vector + 138) with 48. This is risky!

Observe that CPU speed is reduced considerably, (as much as 50% on the 20XX PETs). This will be of no concern if the program

is used solely for enlightenment. You may clear the screen, list the program and even NEW the program and use your own. Just remember the SYS vectors and be sure that the displayed bytes are the ones you want to observe before NEWing.

Please note that when "Front Panel" is enabled, screen area available to the user is reduced to 16 lines (20 on the 80XX). If your program's format relies on a full screen then some undesired scrolling will probably result.

```
1 REM PET/CBM FRONT PANEL BY BOYD RAY
2 REM MUST BE TOTAL OF 16 DATA ITEMS
3 REM ON THREE LINES THAT FOLLOW!
4 DATA5,$E811,142,$E812,143
5 DATA$E844,166,$E845,168,59464
6 DATA$0036,59465,198,59466,216,59467
7 REM IF USING 'OLD ROMS', CHANGE DATA
8 REM 5 TO 90,142 TO 513,143 TO 514,
9 REM 166 TO 547,168 TO 549, $0036 TO
10 REM 136,198 TO 226,AND 216 TO 245
11 REM ON DATA LINES ABOVE
14 IFPEEK(1014)=1THEN25
15 PV=1+PEEK(50003):IFPV=161THENPV=3
16 IFPEEK(57344)=76THENPV=4
17 A=134:B=135:IFPV>1THENA=52:B=53
18 POKE1015,PV:POKE1016,A:POKE1017,B
21 HIMEM=PEEK(A)+PEEK(B)*256
22 POKE59468,12:IFPEEK(HI)=120THEN28
23 AD=HIMEM-642:GOSUB145
24 POKEA,L:POKEB,H:CLR
25 PV=PEEK(1015):A=PEEK(1016):B=PEEK(101
   ¬7)
26 HIMEM=PEEK(A)+PEEK(B)*256
27 DISABLE=HIMEM:ENABLE=DISABLE+32
28 IFPEEK(HI)=120THENSYS(DISABLE)
29 A$="ADDRESS 7 DATA 0":S$="     "
30 PRINT"n";A$;S$;:IFPV=4THENPRINTA$;S$;
31 IFPV=4THENPRINTA$;S$;
32 C$="0123456789ABCDEF":PRINTA$
33 AM=HIMEM+179:FORI=0TO15:READ ADDR$
34 ADDR$=RIGHT$("      "+ADDR$,6)
35 AD=VAL(ADDR$):GOSUB146
36 POKEAM+I*4,L:POKEAM+I*4+1,H
37 PRINTADDR$;
39 PRINT">>>>>>>>>>>>>>>";:NEXT
42 PRINT:I=0:IFPEEK(HI)=120THEN160
44 READ BYTE$:IFBYTE$="!"THEN56
46 BYTE=VAL(BYTE$):I$=RIGHT$(BYTE$,1)
48 IFI$<>"*"THEN54
50 AD=HIMEM+BYTE:GOSUB145
52 POKEHIMEM+I,L:I=I+1:POKEHIMEM+I,H
53 I=I+1:GOTO44
54 POKEHIMEM+I,BYTE:I=I+1:GOTO44
56 TA=HI+177:FORI=0TO15:READD:AD=HI+D
57 GOSUB145:POKETA+I*4,L:POKETA+1+I*4,H
58 NEXT:AD=HIMEM+45:GOSUB145
```

```
 59 POKEHI+34,L:POKEHI+39,H
 60 IFPV=4ANDD=589THEN56
 61 IFPV=1THEN160:REM IF 20XX
 62 POKEHI+2,46:POKEHI+156,46:POKEHI+4,
    ¬144
 63 POKEHI+36,144:POKEHI+5,0:POKEHI+10,0
 64 POKEHI+37,0:POKEHI+42,0:POKEHI+9,145
 65 POKEHI+41,145:FORI=47TO49:POKEHI+I,
    ¬234
 66 NEXTI:POKEHI+143,216:POKEHI+151,216
 67 POKEHI+153,93:POKEHI+154,226
 70 IFPV=2THEN160:REM IF 30XX
 71 POKEHI+2,85:POKEHI+7,228:POKEHI+153,
    ¬127
 72 POKEHI+154,224:POKEHI+156,85
 73 POKEHI+157,228
 74 IFPV=3THEN160:REM IF 40XX
 75 POKEHI+145,5:POKEHI+149,5
 76 POKEHI+22,200:POKEHI+29,200
 77 POKEHI+60,200:POKEHI+64,200
 78 AD=HI+441:GOSUB145:POKEHI+25,L
 79 POKEHI+26,H:POKEHI+57,L:POKEHI+58,H
 80 POKEHI+153,103:GOTO160:REM IF 80XX
120 DATA120,169,133,141,25,2,169,230,141
121 DATA26,2,88,96,162,0,189,0,128,157
122 DATA241*,189,180,128,157,421*,232
123 DATA224,180,208,239,120,169,0,141,25
124 DATA2,169,0,141,26,2,88,96,162,0,32
125 DATA158*,189,241*,157,0,128,189,421*
126 DATA157,180,128,232,224,180,208,236
127 DATA162,0,189,177*,133,1,232,189
128 DATA177*,133,2,232,189,177*,141,96*
129 DATA232,189,177*,141,97*,232,173,0,0
130 DATA141,176*,160,0,185,168*,44,176*
131 DATA240,6,32,132*,76,120*,32,137*
132 DATA200,192,8,48,234,224,64,48,196
133 DATA76,142*,169,81,145,1,96,169,87
134 DATA76,134*,165,245,201,9,176,7,105
135 DATA9,133,245,32,219,229,76,133,230
136 DATA173,64,232,73,32,41,32,240,247
137 DATA96,128,64,32,16,8,4,2,1,255,!
138 DATA289,309,329,349,369,389,409,429
139 DATA449,469,489,509,529,549,569,589
140 DATA329,349,369,389,409,429,449,469
141 DATA489,509,529,549,569,589,609,629
145 H=INT(AD/256):L=AD-H*256:RETURN
146 IFMID$(ADDR$,2,1)<>"$"THEN145
147 D=0:FORJ=1TO4:FORK=1TO16
148 IFMID$(AD$,J+2,1)<>MID$(C$,K,
    ¬1)THEN150
149 C=K-1:K=16
150 NEXTK:D=(D+C)*16:NEXTJ:AD=D/16
151 GOTO145
160 SYS(HI+13)
170 IFPEEK(1014)=1THEN200
180 PRINT"SYS(";DI;")DISABLES/SYS(";
190 PRINTEN;")ENABLES":POKE1014,1
200 REM START OF USER PROGRAM
READY.
```

CHAPTER FOUR: Graphics

Lower Case Descension On The Commodore 2022 Printer

W. M. Bunker

The programmability of the printer makes it possible to do just about anything you want with it, from listing a program to plotting. I'm enclosing one example. The lower case letters on the printer are made as shown on the first two lines at the bottom of the listing. This is common on dox matrix printers — my General Electric Terminet 30 at work, far more expensive than the 2022, does this.

On the 2022, if you don't like their lower case letters, you can make your own. These are shown below the original version. The program listed produces the improved letters. For each line to be printed, define it as P$, then GOSUB 5000, and printing as shown will result.

```
100 DIMLN(6):LN(2)=71:LN(3)=74
110 LN(4)=80:LN(5)=81:LN(6)=89
120 DIMPP$(6):SP$=" ":ST$=""
130 OPEN3,4:OPEN5,4,5:OPEN6,4,6
140 DATA0,0,0,64,0,0
150 DATA57,69,69,63,0,0
160 DATA2,1,1,126,0,0
170 DATA127,68,68,56,0,0
180 DATA56,68,68,127,0,0
190 DATA120,5,5,126,0,0
200 FORI=1TO6:PP$(I)=ST$
210 FORJ=1TO6:READA
220 PP$(I)=PP$(I)+CHR$(A)
230 NEXTJ:NEXTI
240 A$="T[DN]HE QUICK BROWN FOX JUMPED"
250 A$=A$+" OVER THE LAZY DOG'S BACK."
260 B$="J[DN]JJ[UP]G[DN]GG[UP]Q[DN]"
270 B$=B$+"QQ[UP]Y[DN]YY[UP]P[DN]PP."
280 PRINT#3,A$:PRINT#3,B$
290 P$=A$:GOSUB330
300 P$=B$:GOSUB330
310 CLOSE3
320 END
330 KL=0:LL=LEN(P$)
```

```
340 PRINT#5,PP$(1):Q$=ST$
350 FORI=1TOLL
360 CC$=MID$(P$,I,1):CC=ASC(CC$)
370 IFCC=145THENKL=0:GOTO460
380 IFCC=17THENKL=1:GOTO460
390 IFKL=0 GOTO460
400 IFCC=71THENQ$=Q$+SP$:GOTO470
410 IFCC=80THENQ$=Q$+SP$:GOTO470
420 IFCC=81THENQ$=Q$+SP$:GOTO470
430 IFCC=89THENQ$=Q$+SP$:GOTO470
440 IF CC <> 74 GOTO 460
450 Q$ = Q$ + CHR$(254): GOTO 470
460 Q$=Q$+CC$
470 NEXT I
480 PRINT#6,CHR$(5)
490 PRINT#3,Q$
500 FORL=2TO6:PRINT#5,PP$(L)
510 Q$=ST$:KL=0
520 FORI=1TOLL
530 CC$=MID$(P$,I,1):CC=ASC(CC$)
540 IFCC=145THENKL=0:GOTO590
550 IFCC=17THENKL=1:GOTO590
560 IFKL=0ORCC<>LN(L)GOTO580
570 Q$=Q$+CHR$(254):GOTO590
580 Q$=Q$+SP$
590 NEXT I
600 PRINT#3,Q$; CHR$(141);
610 NEXT L
620 PRINT#6,CHR$(19)
630 PRINT#3
640 PRINT#6,CHR$(24)
650 RETURN
```

Plotting With The 2022 Printer

John Winn

The special features of the 2022 printer make for some surprising applications. Here's a remarkable and useful one.

In the January/February issue of **COMPUTE!**, Len Lindsay mentioned using the Commodore Model 2022 Tractor Feed Printer for plotting applications. He pointed out that the ability to vary the line spacing in this printer allows a high degree of vertical resolution. In a somewhat unrelated article in the same issue, W. M. Bunker showed how the user-defined character, CHR$(254), could generate attractive lower case letters with descending tails. He also mentioned the plotting possibilities of this printer. This article puts both ideas together — the variable line spacing and the user-defined character — to produce a plotting subroutine of surprisingly high resolution.

The logic behind this subroutine (which is actually a set of subroutines) is to increase resolution from that given by the height of a line and the width of a space to that of a single printer matrix dot. In the way I will describe the subroutine, I will have in mind a graph of computed values of some function, such as Y=SIN (X). In the second part of the article, I will show how data values can be plotted point by point, how bar graphs could be generated, and how more generalized graphics could be done.

Think about how you would graph the function Y=SIN (X) by hand. You would first decide which direction on your graph paper would be in the X direction, and which the Y direction. For us, the X direction will increase down the page as the printer paper advances, and the Y direction will increase across the page, in the direction of the print head motion. Normally, you would hold the graph paper so that X increases from left to right and Y increases from bottom to top. On the printer plot, this orientation is achieved by tilting your head to the right — the plot will be rotated from the usual orientation as it is produced by the printer.

Your next choice in plotting by hand is to pick the minimum and maximum X and Y values to be spanned by the graph. Let's call these values XN, XX, YN, and YX, respectively, in the program. You would then decide how frequently to mark off the two axes

with tick marks, and how big to make the graph (how much of the graph paper page to use). For the printer, we can choose the size of the graph by specifying the number of columns wide for the Y axis (variable NY) and the number of rows long for the X axis (variable NX). The size of the page limits NY to a maximum of about 60 for a standard 8½"×11" page. The X axis can be as long as you want (until you run out of paper!), but at the line spacing used in the program, one has about 120 lines (rows) per page. Tick marks are specified by stating the number of columns per tick (variable YT) and the number of rows per tick (variable XT).

Your final step in making the graph is to choose a starting value for X, a final value for X, and an increment for X. The smaller the increment, the more points you will have to plot. You then generate values for Y=SIN (X) at each X value, and put a point on your graph paper at each (X,Y) coordinate. The heart of the plotting subroutine does exactly the same thing. Given X and Y pairs of values, the subroutine figures out which row and column they lie in, and turns on the dot in the space which corresponds to that (X,Y) pair. On the printer, the spacing between lines is set so that each row is six dots tall. The column to column spacing is also six dots, giving a resolution of about 1/60". (This is comparable to early digital plotters with 0.01" resolution!)

Now look at the subroutines. The first one (lines 60000-60400) is called via GOSUB 60000 once your main program has established the values for scaling and size mentioned above (NY, NX, YT, XT, XN, XX, YN, and YX). First, it opens files for its own communication with the printer. Secondary addresses 0, 5, and 6 are needed to print, set the special character, and set the line spacing, respectively. Variables needed by the subroutine are computed or initialized, the YN and YX values are printed outside the graph, the Y axis is printed, along with tick marks, using graphic symbols, the line spacing is set via PRINT# 12, CHR$(14), and the XN value is printed, setting up the first row for plotting. The first subroutine returns at this point.

Your main program then computes (X,Y) pairs, one at a time, *in increasing order of X*. After each pair is computed, your program goes to the main subroutine via GOSUB 61000. This subroutine first checks to see if the computed point really lies within the graph limits. If not, it simply returns. Lines 61100 to 61130 find out which row (NR%) of matrix dots contain the point, (X,Y). These lines also find out which printer line (NL%) we are dealing with, and how many blank spaces (NB%) precede the one containing (X,Y). The

special character is built up in the array C%. Whenever a dot appears in a new row or column, the current special character is printed (lines 61200 and 61210) without a carriage return/line feed. If we are suddenly on a new line, that condition is sensed (via the variable LL%, standing for "last line") and the X axis borders (at both ends of the graph and with tick marks if required) are printed to finish the line and move on to the next. If it is recognized that several blank lines are required between the last point and the current one, these are generated as well (lines 61220 through 61340 do all this). The subroutine can RETURN to get the next (X,Y) values at several points in this scheme.

Finally, when all the (X,Y) pairs have been computed, your main program calls the termination subroutine via GOSUB 60500. This routine forces a point to be plotted at (X=XX, Y=YX) to guarantee that any remaining blank lines appear at the "end" of the plot. The final Y axis label, XX, is printed out, along with the "bottom" Y axis border (lines 60550-60750). The line spacing is graciously reset to the default value with the PRINT#12, CHR$(24) statement. Finally, YN and YX values are printed once more, the files are closed, and the subroutine returns.

The graph is now complete as far as these subroutines are concerned. Any graph titles or other notations you may want can be printed before plotting (before the GOSUB 60000) or after plotting (on return from the termination subroutine).

A sample driving program is shown to illustrate these instructions. The REM statements point out what is happening step by step. Of course, the values set by INPUT statements could be computed directly in many cases, as long as they are established before plot initialization. The final plot produced by this program is shown as well. Note that the limiting factor in the plot quality is the mechanical error in print head motion. The print head does not return to exactly the same point with each carriage return. That's why the X axis is a bit wiggly, at least on my printer.

The speed at which a plot is produced depends on two factors. The first is the computation time needed to generate (X,Y) points themselves. The second is quite variable and depends on the shape of the graph and the X increment chosen. Each point line containing at least one plotted point must be printed twice — once for the point, without a line feed, and once for the X axis borders, with a carriage return/line feed. If several plotted points occur on the same print line in a variety of print columns, the line will be overprinted several times, since the special character can be used in

only one way in any single PRINT statement. The example graph took about 5 minutes to plot all 600 points.

A few general comments are in order before turning to ways to extend the subroutine. First, integer variables are used in many places to force truncated integer arithmetic without specifying the INT function. An exception occurs in lines 60200, 60600, 61230 and 61310 where INT is used to sense the need to print a tick mark. Also notice the use of the logical OR in line 61150 to turn on the bits that set the special character. (The 2022 instructions describe how to set this character dot-by-dot.) The OR is needed rather than a simple addition in case the same matrix dot is turned on by more than one (X,Y) pair. The OR will not change a bit already set, while addition will, in general.

Since this subroutine package is meant to be used by a variety of programs, it has a self-contained file structure, opening and closing what it needs independently of the main program. For any particular program, this feature could be removed and all files could be controlled by the main program.

Suppose you want to plot rather widely spaced points, instead of closely spaced points along a curve. Widely spaced single dots are difficult to see. Or suppose two or more different sets of data have to be plotted on the same graph. What is needed in both cases is some way to plot larger and different types of symbols instead of points. One way of plotting symbols is discussed below, but it is not fool-proof.

The simplest way to make an arbitrary symbol (a plus, x, square, or whatever) is to begin by defining the dot-to-dot distances, DX and DY, with the statements

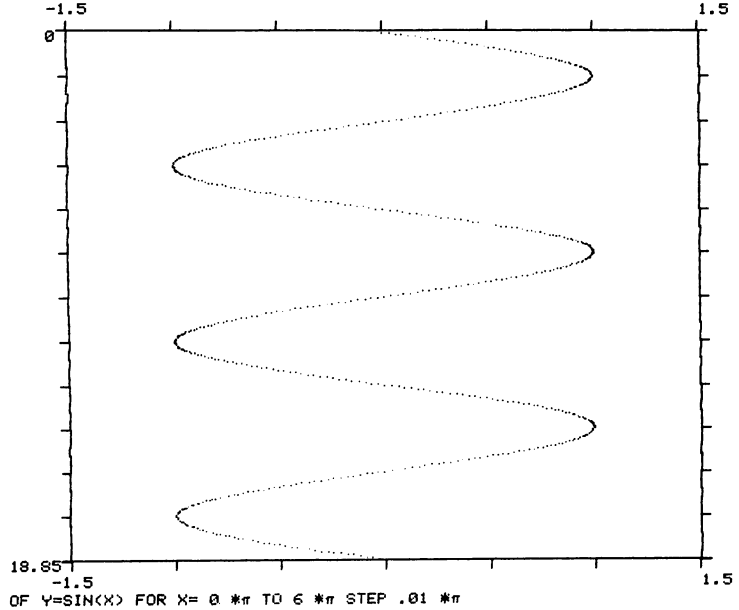
```
DX=(XX-XN) / (6*NX)
DY=(YX-YN) / (6*NY).
```

Then, after computing each (X,Y) pair, call a new subroutine (let's say it is called by GOSUB 62000) in line 340 of the example program instead of the main (61000) subroutine. The new subroutine constructs the symbol you choose, centered about the correct (X,Y) point. In the sample symbol subroutine, the way to plot a small +, three dots wide, is shown. Other symbols can be created in many shapes and sizes, but all are subject to problems if the X values are too closely spaced. If the symbols overlap from one point line to the next, and the next symbol wants to extend back to the previous print line, you have troubles. The printer won't back up. The only safe way around this problem is to go through your data point by point, and create a new data set representing all the

dots in all the symbols for all the original data. This new data has to be arranged in order of increasing X values, but it can then be plotted point-by-point using the original scheme without problems.

More complex graphics — high resolution pictures of arbitrary shapes — will always have to face the problem of one-directional paper motion. A line-by-line "raster scan" is possible, but 60 print columns contain $60 \times 6 \times 6 = 2160$ matrix dots per print line. Turning on the correct dots and scanning all the possibilities could take an intolerable amount of time.

For many applicatins, a bar graph (histogram) is more useful than a point-by-point or line graph. Bar graphs are easily generated using the built-in PET graphics characters, as long as the bars go "across the page" rather than "up and down." The line spacing must be set to the same close spacing used here, or to a closer spacing, if half-size graphic symbols (and lower resolution) are acceptable. The resolution is lower because half-size symbols don't come in as many widths as full size symbols. Even with the built-in graphics symbols, the bar graph won't look perfect, since small gaps in the bar graph silhouette will exist with these symbols. The best approach would use a mixture of built-in graphics plus a special character at the end of the bar to give a continuous silhouette.




```
PLOT OF Y=SIN(X) FOR X= 0 *π TO 6 *π STEP .01 *π
PARAMETERS FOR THIS GRAPH...
NY= 60  NX= 60  YT= 10  XT= 5
SAMPLE PLOT
```

```
60000 OPEN10,4,0:OPEN11,4,5:OPEN12,4,6:IFNY>60THENNY=60
60050 XT%=XT:YT%=YT:LL%=0:LB%=0:NX%=6*NX-1:NY%=6*NY-1
60100 CR$=CHR$(141):SC$=CHR$(254):Q=(NY%+1)/6-LEN(STR$(YN)):IFQ<0THENQ=0
60150 PRINT#10:PRINT#10,TAB(9)YN;TAB(Q)YX:PRINT#10,"          _";:FORQ=1TONY
60200 IFQ/YT%=INT(Q/YT%)THEN60300
60250 PRINT#10,"_";:GOTO60350
60300 PRINT#10,"_";
60350 NEXT:PRINT#10,"_":PRINT#12,CHR$(14):Q$=STR$(XN):Q$=RIGHT$(Q$,LEN(Q$)-1)
60400 PRINT#10,TAB(10-LEN(Q$))Q$;CR$;:RETURN
```

INITIALIZATION SUBROUTINE

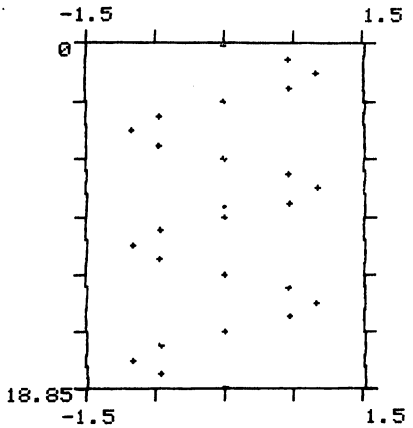
```
100 REM INITIALIZE PLOT SIZE VARIABLES
110 INPUT "HOW MANY COLUMNS (1 TO ¬
    ¬60)";NY
120 INPUT "HOW MANY COLUMNS PER TICK";YT
130 INPUT "HOW MANY ROWS (120 PER ¬
    ¬PAGE)";NX
140 INPUT "HOW MANY ROWS PER TICK";XT
150 INPUT "X MIN AND X MAX";XN,XX
160 INPUT "Y MIN AND Y MAX";YN,YX
170 INPUT "X START AND END (UNITS OF   ¬
    ¬)";X1,X2
180 INPUT "X INCREMENT (UNITS OF   )";XS
200 REM CALL INITIALIZATION SUBROUTINE
210 GOSUB 60000
300 REM DEFINE THE FUNCTION TO BE ¬
    ¬PLOTTED POINT BY POINT
310 FOR X=X1*   TO X2*   STEP XS*
320 Y=SIN(X)
330 REM CALL MAIN SUBROUTINE FOR EACH ¬
    ¬POINT
340 GOSUB 61000
350 NEXT
400 REM CALL THE PLOT TERMINATION ¬
    ¬SUBROUTINE
410 GOSUB 60500
500 REM ADD A TITLE TO THE BOTTOM OF ¬
    ¬THE GRAPH
510 OPEN1,4,0
520 PRINT#1,"PLOT OF Y=SIN(X) FOR ¬
    ¬X="X1"*   TO "X2"*   STEP "XS"*  "
600 REM PRINT OUT REMAINING GRAPH ¬
    ¬PARAMETERS
610 PRINT #1,"PARAMETERS FOR THIS GRAPH ¬
    ¬..."
620 PRINT#1,"NY="NY" NX="NX" YT="YT" ¬
    ¬XT="XT
999 END
```

SAMPLE DRIVING PROGRAM

```
61000 IF (Y<YN)OR(Y>YX)OR(X<XN)OR(X>XX) ¬
      ¬THEN RETURN
61100 NC% = 1+(Y-YN)/(YX-YN)*NY%
61110 NB% = (NC%-1)/6 : NC% = NC%-6*NB%
61120 NR% = 1+(X-XN)/(XX-XN)*NX%
61130 NL% = (NR%-1)/6 : NR% = NR%-6*NL%
61140 IF NL% <> LL% OR NB% <> LB% THEN ¬
      ¬GOSUB 61200
61150 LL%=NL% :LB%=NB% :C%(NC%)=C%(NC%) ¬
      ¬OR 2^(6-NR%) : RETURN
61200 Q$ = "" : FOR Q=1 TO 6 :Q$=Q$+CHR$
      ¬(C%(Q)) : C%(Q)=0 : NEXT :
      ¬ PRINT#11,Q$
61210 PRINT#10,TAB(LB%+11)SC$CR$
61220 IF NL%=LL% THEN RETURN
61230 IF (LL%+1)/XT% = INT((LL%+1)/XT%) ¬
      ¬THEN 61250
61240 PRINT#10,TAB(10)"Y" TAB(NY)"T" :
```

```
      ¬ GOTO 61260
61250 PRINT#10,TAB(10) " " TAB(NY) "L"
61260 IF NL% <> LL%+1 AND NL% <> LL% ¬
      ¬THEN GOSUB 61300
61270 RETURN
61300 FOR Q=1 TO NL%-LL%-1
61310 IF (LL%+Q+1)/XT%=INT((LL%+Q+1)/   ¬
      ¬XT%) THEN 61330
61320 PRINT#10, TAB(10) "T" TAB(NY)   ¬
      ¬"T" : GOTO 61340
61330 PRINT#10, TAB(10) " " TAB(NY) ¬
      ¬"L"LIST
61340 NEXT : RETURN
MAIN PLOTTING SUBROUTINE
```

Note: These listings were done with a CBM 32B computer which has a slightly different graphics character set and no key for PI. In the Sample Driving Program, Lines 170, 180, 310, and 520 should have the symbol for PI inserted in the spaces left blank. In the Main Plotting Subroutine, lines 61250 and 61330 have a blank in quotes which should be :, i.e. the "opposite" graphics character to the L.




```
PLOT OF Y=SIN(X) FOR X= 0 *π TO 6 *π STEP .25 *π
PARAMETERS FOR THIS GRAPH...
NY= 20  NX= 30  YT= 5  XT= 5
```

SAMPLE SYMBOL PLOT

```
60500 X=XX:Y=YX:GOSUB61000:GOSUB61200:GOSUB61250
60550 Q$=STR$(XX):Q$=RIGHT$(Q$,LEN(Q$)-1):PRINT#10,TAB(10-LEN(Q$))Q$"T";
60600 FORQ=1TONY:IFQ/YT%=INT(Q/YT%)THEN60700
60650 PRINT#10,"-";:GOTO60750
60700 PRINT#10,"T";
60750 NEXT:PRINT#10,"-":PRINT#12,CHR$(24)
60800 Q=(NY%+1)/6-LEN(STR$(YN)):IFQ<0THENQ=0
60850 PRINT#10,TAB(9)YN;TAB(Q)YX
60900 CLOSE10:CLOSE11:CLOSE12:RETURN

PLOT TERMINATION SUBROUTINE
```

```
 62000 REM ALTERNATE SUBROUTINE FOR PLOTTING DATA AS A SMALL +
 62010 REM DX AND DY MUST HAVE BEEN DEFINED BY THE MAIN PROGRAM
 62020 X=X-DX:GOSUB 61000
 62030 X=X+DX:Y=Y-DY:GOSUB 61000
 62040 Y=Y+DY:GOSUB 61000
 62050 Y=Y+DY:GOSUB 61000
 62060 Y=Y-DY:X=X+DX:GOSUB 61000
 62070 REM RESTORE X TO ORIGINAL VALUE BEFORE RETURNING
 62080 X=X-DX
 62090 RETURN
READY.
SAMPLE SYMBOL SUBROUTINE
```

Keyprint

Charles Brannon

KEYPRINT is an easy solution to many hardcopy problems. For example, how would you copy the instructions from a computer game onto your printer? The obvious solution is to modify the program to direct its output to the printer. This is, however, time-consuming. Besides, what if — horror of horrors — you do not know how to make this modification?

So what does KEYPRINT do, anyway? Simple. You just touch a single key and the entire screen is copied onto the printer. This can happen at any time: while calculations are in progress, during a game of STARTREK, after a print-out of information to the screen, when you touch that certain key accidentally — *anytime*. KEYPRINT totally interrupts everything PET is doing, dumps the screen onto the printer, and then returns control back to BASIC as though nothing had happened.

KEYPRINT's uses are multitudinous. No longer do you have to write special printer subroutines. It's just touch and go. Your software can even call the screen dump directly with a SYS command. If you have a Commodore 2022 printer, you can copy graphics verbatim. (Remember to set the lines-per-inch to eight first. A side-effect of this is that text looks crammed together; remember to reset the lpi to six.) So here's how to use KEYPRINT:

1. Enter the machine language monitor with an SYS 1024 command.
2. If you've already typed in and saved KEYPRINT, enter:

   .L "KEYPRINT",01

and hit RETURN. Now type an X, hit RETURN and go to step 6.

3. Otherwise, list the block of memory that KEYPRINT occupies with:

   .M 033A 03CB

4. Now, using the cursor, replace the "numbers" (which often contain alphabetic characters, since they're hexadecimal) with the one shown in the listing. Type these bytes in *exactly* as shown. (All machine language program instructions seem to stress that, but it's really important as the program will CRASH if you don't type it in perfectly right.) Remember to hit RETURN after each line.
5. Save the program by entering:

   .S "KEYPRINT",01,033A,03CB (Afterwards, enter .X to exit to BASIC)

6. Now activate KEYPRINT with:

SYS 826 (hit RETURN)

The cursor should re-appear almost instantly, blinking merrily under READY. If it doesn't, then your PET has *crashed.* Why? Either you typed in the program incorrectly, (even one tiny mistake) or you're using an original ROM PET. Shame on you! Go back and check over that program you typed in! (Aren't you glad you saved it first?)

7. Hopefully, your cursor came back. That means that KEYPRINT is ready and rarin' to go. How do you make it work? Just press the " \" key. If you have a printer hooked up that responds to a device number of 4, then the entire screen will be printed onto your printer. For devices other than 4, POKE 858, DN where DN equals the device number of your printer. (If the above terminology seems confusing, don't worry. If you have a Commodore printer, everything will work fine. If not, then I can't guarantee flawless operation.)

8. KEYPRINT remains in your machine until you turn it off or you otherwise interrupt its power supply (Like dropping the PET or setting it on fire). KEYPRINT can be de-activated, however, by a simple procedure: Hold down the shift key and press the RUN/STOP key or type in LOAD and hit RETURN. Ignore any messages the PET gives. Now press the RUN/STOP key again. The word BREAK is displayed and that is exactly what you did to KEYPRINT — you broke it. It will work no longer. IMPORTANT NOTE: loading any program also "breaks" KEYPRINT. In either case, you can re-activate it with a SYS 826.

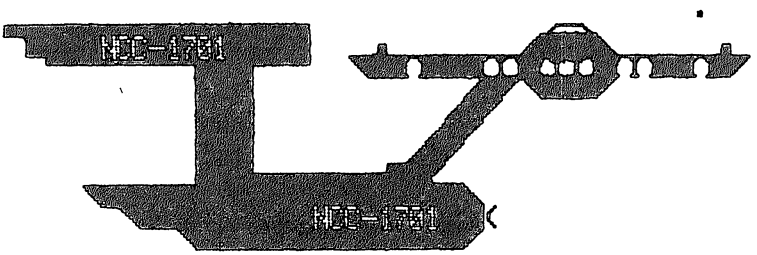
9. If you don't want to have to type a key to dump the screen, use a SYS 849 either in direct mode or within a program. It does not matter whether KEYPRINT is "activated" or not for the command to work.

So there you have it. I plan to use KEYPRINT quite a bit in the future. I think of it as a "Wedge" for the printer as DOS SUPPORT (Commodore) is for the 2040.

P.S. I want to make it clear which key is used to print the screen: it is the key to the right of the ampersand at the top of the keyboard, not the shift of "M." Also, be aware that some programs use the second cassette buffer (where KEYPRINT resides) for data storage or for their own machine language programs.

Resources:
Butterfield, Jim. "PET in Transition" **COMPUTE!**, pp 68-70 (Fall, 1979)
Sheward, D. "Listing from Commodore's 'The Transactor' ",
The Paper, p. 39 (March/April 1980)

```
B*
      PC   IRQ   SR AC XR YR SP
.;   0401 0345 32 04 5E 00 F4
.M 033A 03CB
.:   033A 78 A9 03 85 91 A9 45 85
.:   0342 90 58 60 A5 97 C9 45 D0
.:   034A 03 20 51 03 4C 2E E6 A9
.:   0352 80 85 20 A9 00 85 1F A9
.:   035A 04 85 B0 85 D4 20 BA F0
.:   0362 20 2D F1 A9 19 85 22 A9
.:   036A 0D 85 21 20 D2 FF A9 11
.:   0372 AE 4C E8 E0 0C D0 02 A9
.:   037A 91 20 D2 FF A0 00 B1 1F
.:   0382 29 7F AA B1 1F 45 21 10
.:   038A 0B B1 1F 85 21 29 80 49
.:   0392 92 20 D2 FF 8A C9 20 B0
.:   039A 04 09 40 D0 0E C9 40 90
.:   03A2 0A C9 60 B0 04 09 80 D0
.:   03AA 02 49 C0 20 D2 FF C8 C0
.:   03B2 28 90 CB A5 1F 69 27 85
.:   03BA 1F 90 02 E6 20 C6 22 D0
.:   03C2 A6 A9 0D 20 D2 FF 4C CC
.:   03CA FF 72 21 61 3F 7F 76 57
.X\
```




```
YOU COMMAND THE STARSHIP ENTERPRISE.
THE OBJECT OF THE GAME IS TO DESTROY
ALL THE ENEMY KLINGONS IN THE GALAXY
BEFORE YOUR TIME IS UP. AT THE START
OF EACH GAME PET TELLS YOU
   HOW MANY KLINGONS YOU MUST DESTROY
   HOW MUCH TIME YOU STARDATES YOU HAVE
   AND HOW MANY STARBASES THERE ARE.
```

80 x 50 Graphics

Murray D. Weingarten

One of the deficiencies in BASIC, as written for the PET, is that it does not allow the user to set or clear on the screen an individual point specified by two coordinates. Hence, to set, for example, the point in row 7, column 15 of the screen, one needs to write statements of the form

```
10 A=32768+7*40+15 : POKE A, 160
```

instead of simply SET(15,7). Even then, the maximum resolution obtainable using this approach is 40×25.

With a little extra programming, it is possible to overcome these difficulties. The PET graphics character set includes all the characters necessary to support a screen resolution of 80×50. The approach to use was laid out in a program written by Jerry Velders in the PET USER NOTES (vol. 2, #1) in 1979. That program, however, did not allow the user to clear points, only to set them. Moreover, the coordinates of the point to be set had to be POKEd into zero page memory, adding unnecessary BASIC statements.

The program listed here POKEs a machine language routine anywhere in the RAM space of the PET. Once it has been RUN, the coordinates of a point on the screen may be specified in any BASIC statement by the variable names XX and YY ($0 \leq XX \leq 79$; $0 \leq YY \leq 49$). Executing the function USR(1) will then set that point on the screen, while USR(0) will clear the point. It is up to the user to ensure that XX and YY contain legal coordinate values before the USR function is executed.

Following are some simple programming examples:

(1)
```
10 PRINT "clr"
20 FOR YY=0 TO 49: XX=0: A= USR(1): XX=79:
   A=USR(1): NEXTYY
30 FOR XX=0 TO 79: YY=0: A= USR(1): YY=49:
   A=USR(1): NEXTXX
```

puts a border around the screen.

(2)
```
10 PRINT "clr"
20 FOR XX =0 TO 79
30 YY=25 — 24*SIN(XX/5): A=USR(1): NEXTXX
```

plots a sine wave

(3)
```
10 PRINT "clr"
20 X1=1:XX=0:YY=5
30 IF XX>79 OR XX<0 THEN X1=X1: GOTO 50
```

```
40 A=USR(1)
50 FOR A=0 TO 10: NEXTA: A=USR(0):XX=XX+X1:
   GOTO 30
```

moves a point back and forth across the screen.

The disassembly shows how the program looks when POKEd into high memory on an 8K PET. The code from 1EDB to 1F4B extracts the values of XX and YY from the BASIC variables storage area of memory. The code from 1F4D to the end is analogous to Jerry Velders' program. It calculates the correct screen address, determines what is already at that screen address, and POKEs that address with the necessary graphics character.

I have found the routine to be particularly useful in the graphic display of numerical data on the screen, where doubling the resolution of the screen is more important than having available a wide variety of graphics symbols.

The routine, as written, will only operate on original ROM PETs. This is due to the use of the BASIC subroutines FLPINT, ABS, and MVFACC, and the use of the pointers at 7C to 7F to point to BASIC variables storage. Substitution of the correct addresses should allow it to work with other ROMs as well.

```
1EC9    20 7C 6C E1 7B FF 62 FE
1ED1    7E E2 7F FB 61 EC FC A0   ! TABLE.
1EDB    08          PHP
1EDC    D8          CLD
1EDD    20 A7 D0    JSR FLPINT    Save parameter (indicating
1EE0    A5 B4       LDA B4        SET or CLEAR)
1EE2    8D C5 1E    STA 1EC5      at 1EC5.
1EE5    A9 02       LDA #02       Counter = 2.
1EE7    8D C8 1E    STA 1EC8
1EEA    A5 7C       LDA 7C        Save contents of 7C
1EEC    8D C6 1E    STA 1EC6      and 7D (start of BASIC
1EEF    A5 7D       LDA 7D        variables).
1EF1    8D C7 1E    STA 1EC7
1EF4    A0 00       LDY #00       Load acc. with first
1EF6    B1 7C       LDA (7C),Y    letter of a variable
1EF8    C9 58       CMP #58       name.
1EFA    F0 22       BEQ 1F1E
1EFC    C9 59       CMP #59       Is it 'X' or 'Y'?
1EFE    FO 1E       BEQ 1F1E
1F00    A5 7C       LDA 7C        NO. Increase pointer
1F02    18          CLC           at 7C by 7.
1F03    69 07       ADC #07
1F05    90 02       BCC 1F09
1F07    E6 7D       INC 7D
1F09    85 7C       STA 7C
1F0B    38          SEC           Have we gone
1F0C    A5 7F       LDA 7F        too far?
1F0E    E5 7D       SBC 7C
```

```
1F10    10 E2      BPL 1EF4
1F12    AD C6 1E   LDA 1EC6
1F15    85 7C      STA 7C
1F17    AD C7 1E   LDA 1EC7
1F1A    85 7D      STA 7D
1F1C    28         PLP
1F1D    60         RTS
1F1E    C8         INY
1F1F    D1 7C      CMP (7C),Y
1F21    DO DD      BNE 1F00
1F23    48         PHA
1F24    A4 7D      LDY 7D
1F26    A5 7C      LDA 7C
1F28    18         CLC
1F29    69 02      ADC #02
1F2B    90 01      BCC 1F2E
1F2D    C8         INY
1F2E    20 74 DA   JSR MVAFACC
1F31    20 2A DB   JSR ABS
1F34    20 A7 D0   JSR FLPINT
1F37    68         PLA
1F38    AA         TAX
1F39    A5 B4      LDA B4
1F3B    E0 58      CPX #58
1F3D    F0 06      BEQ 1F45
1F3F    8D C1 1E   STA 1EC1
1F42    4C 48 1F   JMP 1F48
1F45    8D C0 1E   STA 1EC0
1F48    CE C8 1E   DEC 1EC8
1F4B    D0 B3      BNE 1F00

1F4D    A2 02      LDX #02
1F4F    18         CLC
1F50    BD BF 1E   LDA 1EBF,X
1F53    09 FE      ORA #FE
1F55    69 01      ADC #01
1F57    9D C2 1E   STA 1EC2,X
1F5A    CA         DEX
1F5B    D0 F2      BNE 1F4F
1F5D    AD C3 1E   LDA 1EC3
1F60    CD C4 1E   CMP 1EC4
1F63    D0 0D      BNE 1F72
1F65    AD C3 1E   LDA 1EC3
1F68    D0 15      BNE 1F7F
1F6A    A9 02      LDA #02
1F6C    8D C2 1E   STA 1EC2
1F6F    4C 8C 1F   JMP 1F8C
1F72    AD C3 1E   LDA 1EC3
1F75    D0 10      BNE 1F87
1F77    A9 01      LDA #01
1F79    8D C2 1E   STA 1EC2
1F7C    4C 8C 1F   JMP 1F8C
1F7F    A9 08      LDA #08
1F81    8D C2 1E   STA 1EC2
1F84    4C 8C 1F   JMP 1F8C
```

NO. next variable.

YES. restore 7C and 7D, and return to BASIC.

YES. Is next letter of variable name the same? no. next variable.

yes. set Y and acc. to point to variable.

move to FACC
ensure positive.
convert to integer.

store at 1EC0 (XX) or 1EC1 (YY).

go back for next coordinate.
both XX and YY obtained.

For XX and YY, store 255 if even or 0 if odd.

Determine quadrant number pointed to by XX and YY.

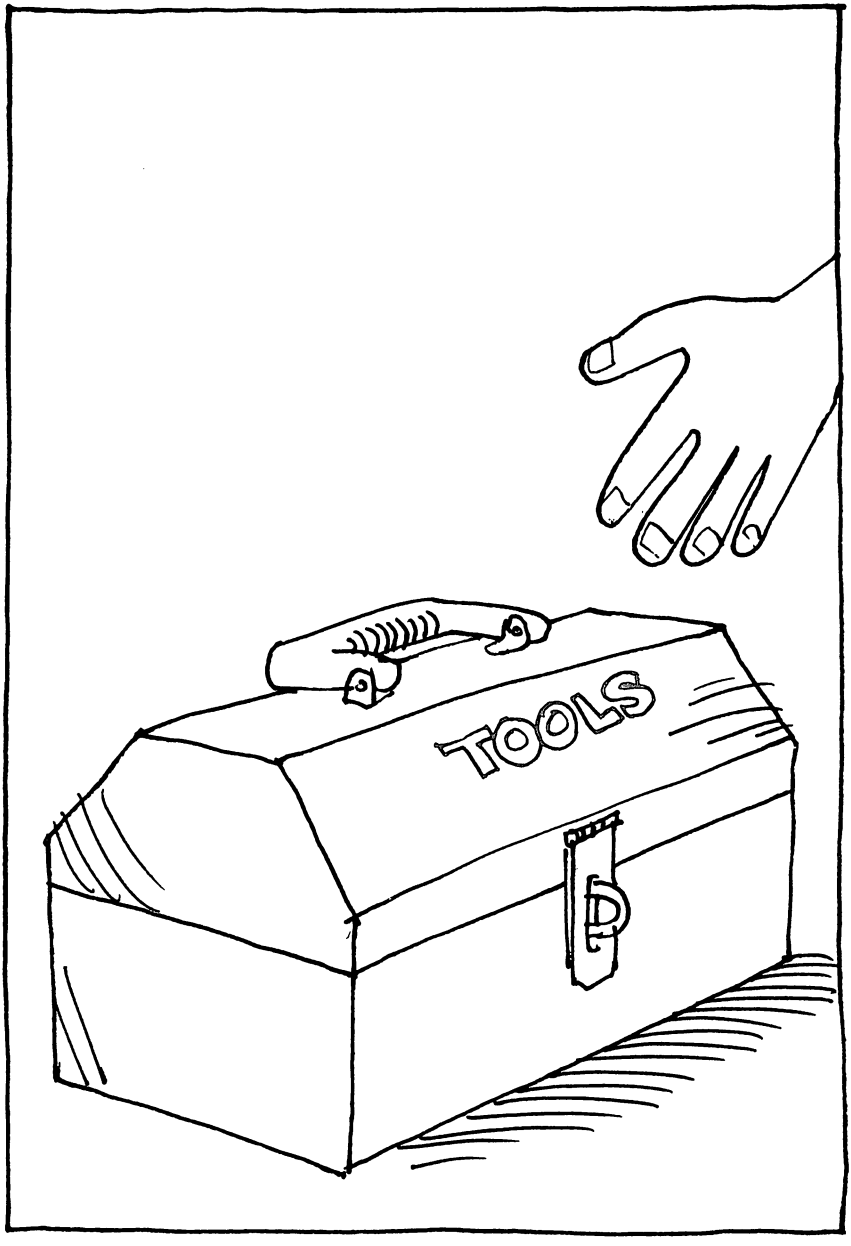
8	1
4	2

```
1F87    A9 04     LDA #04
1F89    8D C2 1E  STA 1EC2

1F8C    4E C0 1E  LSR 1EC0      Calculate screen address
1F8F    4E C1 1E  LSR 1EC1      pointed to by XX and YY.
1F92    A9 D8     LDA #D8
1F94    8D BD 1F  STA 1FBD      1) Divide XX and YY
1F97    A9 7F     LDA #7F          by 2.
1F99    8D BE 1F  STA 1FBE
1F9C    AE C1 1E  LDX 1EC1      2) Calculate 32768 +
1F9F    E8        INX              40*YY+XX
1FA0    18        CLC
1FA1    AD BD 1F  LDA 1FBD      3) Store address at
1FA4    69 28     ADC #28          1FBD and 1FBE.
1FA6    90 04     BCC 1FAC
1FA8    18        CLC
1FA9    EE BE 1F  INC 1FBE
1FAC    CA        DEX
1FAD    D0 F5     BNE 1FA4
1FAF    6D C0 1E  ADC 1EC0
1FB2    90 03     BCC 1FB7
1FB4    EE BE 1F  INC 1FBE
1FB7    8D BD 1F  STA 1FBD

1FBA    A2 00     LDX #00       Load acc. from screen
1FBC    AD __ __  LDA _____     address, and find
1FBF    DD C9 1E  CMP 1EC9,X    corresponding value in
1FC2    F0 08     BEQ 1FCC      table at 1EC9.
1FC4    E8        INX           Return with no action
1FC5    E0 10     CPX #10       if match not found.
1FC7    D0 F6     BNE 1FBF
1FC9    4C 12 1F  JMP 1F12

1FCC    AD BD 1F  LDA 1FBD      Transfer screen address
1FCF    8D F9 1F  STA 1FF9      at 1FBD and 1FBE, to
1FD2    AD BE 1F  LDA 1FBE      1FF9 and 1FFA.
1FD5    8D FA 1F  STA 1FFA
1FD8    AC C5 1E  LDY 1EC5      SET OR CLEAR?
1FDB    F0 0B     BEQ 1FE8

1FDD    8A        TXA           SET.
1FDE    0D C2 1E  0RA 1EC2      OR table position.
1FE1    AA        TAX           with quadrant number.
1FE2    BD C9 1E  LDA 1EC9,X
1FE5    4C F8 1F  JMP 1FF8

1FE8    AD C2 1E  LDA 1EC2      CLEAR.
1FEB    49 0F     E0R #0F       AND table position
1FED    8D C2 1E  STA 1EC2      with complement of
1FF0    8A        TXA           quadrant number.
1FF1    2D C2 1E  AND 1EC2
1FF4    AA        TAX
1FF5    BD C9 1E  LDA 1EC9,X
1FF8    8D __ __  STA _____     POKE screen & return.
1FFB    4C 12 1F  JMP 1F12
```

```
10 PRINT"{CLEAR}WHERE WOULD YOU LIKE T
   HIS ROUTINE LOADED (0=HIGH MEM
   ORY)"
110 INPUTB:IFB<>0THENPRINT"{CLEAR}":B1=
   B:GOTO410
210 B=256*PEEK(135)+PEEK(134)-320
310 POKE135,INT((B-1)/256):POKE134,B-1-
   256*INT((B-1)/256):CLR
320 B1=256*PEEK(135)+PEEK(134)+1:B=B1
350 PRINT"{CLEAR}"
410 PRINT"{HOME}";B:READN$:IFLEFT$(N$,1
   )="X"THENN=VAL(RIGHT$(N$,LEN(N
   $)-1)):GOTO520
450 IFN$="END"THEN3000
510 GOSUB610:GOTO410
520 AH=INT((B1+N)/256):AL=B1+N-256*AH
530 POKEB,AL:POKEB+1,AH:B=B+2:GOTO410
610 R$=RIGHT$(N$,1):R=ASC(R$):L$=LEFT$(
   N$,1):L=ASC(L$)
710 IFR<65THENR=R-48:GOTO910
810 R=R-55
910 IFL<65THENL=L-48:GOTO1110
1010 L=L-55
1110 L=L*16+R:POKEB,L:B=B+1:RETURN
2000 DATA00,00,00,00,00,00,00,00,00
2010 DATA20,7C,6C,E1,7B,FF,62,FE,7E,E2
2020 DATA7F,FB,61,EC,FC,A0,00,00
2030 DATA08,D8,20,A7,D0,A5,B4,8D,X05
2040 DATAA9,02,8D,X08,A5,7C,8D,X06,A5
2050 DATA7D,8D,X07,A0,00,B1,7C,C9,58,F0
2060 DATA22,C9,59,F0,1E,A5,7C,18,69,07
2070 DATA90,02,E6,7D,85,7C,38,A5,7F,E5
2080 DATA7D,10,E2,AD,X06,85,7C,AD,X07
2090 DATA85,7D,28,60,C8,D1,7C,D0,DD,48
2100 DATAA4,7D,A5,7C,18,69,02,90,01
2110 DATAC8,20,74,DA,20,2A,DB,20,A7,D0
2120 DATA68,AA,A5,B4,E0,58,F0,06,8D,X01,
   4C
2130 DATAX136,8D,X00,CE,X08,D0,B3
2140 DATAA2,02,18,BD,X-01,09,FE,69,01,9D
   ,X02
2150 DATACA,D0,F2,AD,X03,CD,X04,D0,0D
2160 DATAAD,X03,D0,15,A9,02,8D,X02,4C
2170 DATAX204,AD,X03,D0,10,A9,01,8D,X02
2180 DATA4C,X204,A9,08,8D,X02,4C,X204
2190 DATAA9,04,8D,X02,4E,X00,4E,X01
2200 DATAA9,D8,8D,X253,A9,7F,8D,X254
2210 DATAAE,X01,E8,18,AD,X253,69,28,90
2220 DATA04,18,EE,X254,CA,D0,F5,6D,X00
2230 DATA90,03,EE,X254,8D,X253,A2,00
```

```
2240 DATAAD,00,00,DD,X09,F0,08,E8,E0,10
2250 DATAD0,F6,4C,X82,AD,X253,8D,X313
2260 DATAAD,X254,8D,X314,AC,X05,F0,0B
2270 DATA8A,0D,X02,AA,BD,X09,4C,X312
2280 DATAAD,X02,49,0F,8D,X02,8A,2D,X02
2290 DATAAA,BD,X09,8D,00,00,4C,X82,END
3000 BH=INT((B1+27)/256):BL=B1+27-256*BH
    :POKE1,BL:POKE2,BH
4000 PRINT"{02 DOWN}{REV}80 X 50 GRAPHIC
    S ROUTINE{OFF}"
4010 PRINT"{02 DOWN}THIS ROUTINE OCCUPIE
    S";B-B1;"BYTES AT"
4020 PRINT"LOCATION";B1
4030 PRINT"{DOWN}RESERVE VARIABLE NAMES ~
    'XX' AND 'YY'"
4040 PRINT"FOR THE COORDINATES OF THE PO
    INT TO BE"
4045 PRINT"SET OR CLEARED."
4050 PRINT"{02 DOWN}XX= 0 TO 79       YY
    =0TO49"
4060 PRINT"{02 DOWN}{REV}SET{OFF} A POIN
    T WITH 'USR(1)'"
4070 PRINT"{REV}CLEAR{OFF} A POINT WITH ~
    'USR(0)'"
4080 PRINT"{DOWN}FOR PROGRAMMING EXAMPLE
    , TYPE 'RUN5000'"
4090 END
5000 X=40:Y=25:DX=RND(-TI):PRINT"{CLEAR}
    ":G=1
5010 DT=3+G*(1+INT(2*RND(1))):ONDTGOTO50
    15,5020,5022,5025,5030
5015 DX=1:DY=0:GOTO5040
5020 DX=-1:DY=0:GOTO5040
5025 DX=0:DY=-1:GOTO5040
5030 DX=0:DY=1:GOTO5040
5040 L=2+INT(30*RND(1))
5050 FORM=1TOL
5060 X=X+DX:Y=Y+DY:IFX=-1THENX=79:GOTO51
    00
5070 IFX=80THENX=0:GOTO5100
5080 IFY=50THENY=0:GOTO5100
5090 IFY=-1THENY=49:GOTO5100
5100 IFDX<>0THENYY=Y:XX=X:A=USR(1):YY=Y-
    1:A=USR(0):YY=Y+1:A=USR(0):GOT
    O5150
5120 YY=Y:XX=X:A=USR(1):XX=X-1:A=USR(0):
    XX=X+1:A=USR(0)
5150 NEXTM
5160 G=-G:GOTO5010
```

CHAPTER FIVE:
Utilities



Cross-Reference For The PET

Jim Butterfield

One of the handy things about the 2040 disk system is that it allows you to read programs — or write them, for that matter — as if they were data files.

The possibilities are endless: you can analyze or cross-reference programs; renumber them; repack them into the minimum number of lines, deleting spaces, comments, etc.; or even create a program-writing program that is tailor-made for a particular job.

This program does cross-referencing of a BASIC program. It's written in BASIC: that means that it won't run too fast (all those GET statements), but you can read what it's doing fairly easily.

There are two types of cross-references normally needed for a BASIC program. One is the variable cross-reference: where do I use B$? The other is a line number cross-reference: when do I go to line 360? This program does either. An example of both types is shown — the program in this case did the cross-references of itself.

Reading a BASIC Program as a File.

To read a BASIC program, you must open it as a file, using type P for program. Line 170 of the cross-reference program does this.

If you read a zero character from the program (that's CHR$(0), not ASCII zero which has a binary value of 48), the GET command gives you a small problem: it will give you a null string instead of the CHR$(0) you might normally expect. You need to watch for this condition and correct it where necessary: you'll see this type of coding in lines 260, 270, and 300.

The first thing to do when you open the file is to get the first two bytes. These represent the program start address, and should be CHR$(1) and CHR$(4) for a normal BASIC program starting at hexadecimal 0401. (See line 180).

Now you're ready to start work on a line of BASIC. The first two bytes are the forward chain. If they are both zero (null string) we have reached the end of the BASIC program; otherwise, we don't need them for this job. (See line 240).

Continuing on the BASIC line: the next pair of bytes represent the line number, coded in binary. We're likely to need this, so we calculate it as L (lines 260 to 280) and also create its

string equivalent. L$. We take an extra moment to right-justify the string by putting spaces at the front so that it will sort into proper numeric order.

From this point on, we are looking at the text of the BASIC line until we reach a zero which flags end-of-line. At that time, we go back and grab the next line.

Detailed Syntax Analysis.

When digging out variables or line numbers, we have several jobs to do. As we look through BASIC text, we must find out where the variable or line number starts. For a variable, that's an alphabetic character; for a line number, it's the preceding keyword GOTO, GOSUB, THEN or RUN followed by an ASCII numeric.

Once we've "acquired" the variable or line number, we must pick up its following characters and tack them on. For line numbers, it's strictly numeric digits. For variables, things are more complex. Both alphabetic and numeric digits are allowed, but we should throw away all after the first two, since GRUMP and GROAN are the same variable (GR) in PET Basic. We must also pick up a type identifier — % for integer variables or $ for strings — if present. Finally, we have to spot the left bracket that tells us we have an array variable.

To help us do this rather complex job, we construct a character type table. Each entry in the table represents an ASCII character, and classifies it according to its type. Numeric characters are type 6. If we're looking for variables, alphabetic characters are type 5, identifiers (% and $) are type 7, and the left bracket is type 8.

To help us in scanning the BASIC line, we define the end-of-line character as type 0; the quotation mark as type 2; the REM token as type 3; and the DATA token as type 4.

Every time we get a new character from BASIC, we get its type from table C as variable C9. If we're looking for a new variable or line number, we see if it matches C — alphabetic for variables, numeric for line numbers. Once we find the new item, we kick C out of range and start searching based on the value of C1. This mechanism means that we can search for a variable starting with an alphabetic, and then allow the variable to continue with alphabetics, numerics, or whatever.

To summarize variables in this area: A is the identity of the character we have obtained from the BASIC program, and C9 is its type. If we're searching, C is the type we are looking for; otherwise it's kicked out of range, to -1 or 9. C1 tells us we're collecting

characters and what types we're allowed to collect. C2 is our variables/line numbers flag; it tells us what we're looking for. M$ is the string we've assembled.

The routine from 480 to 520 scans ahead to skip over strings in quotes and DATA and REM statements.

Collecting The Results.

For each of the BASIC programs we are analyzing, we collect and sort any items we find, eliminating duplicates. They are staged in array A$ in lines 320 to 370. If they are line numbers, they will be left justified so that the sort will be a little odd — line 100 will come before line 20 since we use a string comparison.

When we're ready to start a new line, we add this table to our main results table, array X$, in lines 200 to 220. To save sorting time, we merge these pre-sorted values into the main table. At this point, our data has the line number stuck on the end; this way, we're handling two values with a single array.

Because the merging of the two tables must start at the top so that we can make room for the new items, the items are handled in reverse alphabetic order. We print this to the screen so that you can watch things working. At BASIC speed, this program can take quite a while to run; it's nice to confirm that the computer is doing something during this period.

Final Output.

We finish the job starting at line 530. It's mostly a question of breaking the stuck-together strings apart again and then checking to see if we need to start a new line.

Do Your Own Thing.

The size of array X$ determines how large a program you can handle. The given value of 500 is about right for 16K machines; on 32K you can raise it to 1500 or so.

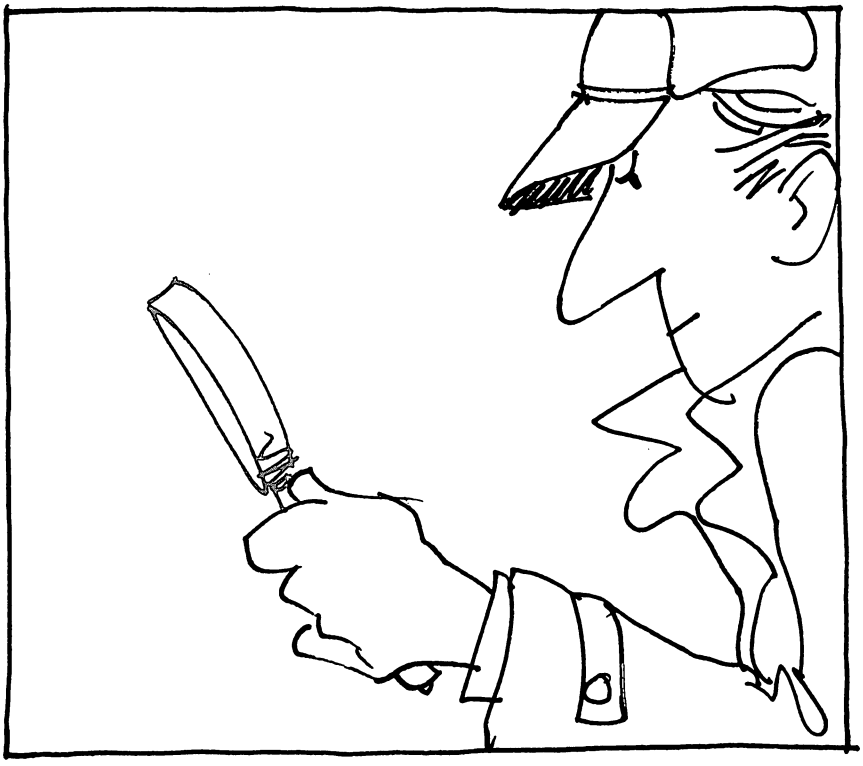
If you're squeezed for space, change array C to an integer array C%. As you can see from the cross-reference listing, you'll need to change lines 100, 140, 150, 160, and 310 — see how handy the program is?

As mentioned before, run time is slow. A machine language program — or even a BASIC program with machine language inserts — would speed things up dramatically.

```
100 DIM A$(15),B$(3),X$(500),C(255)
110 PRINT"CROSS-REF    JIM BUTTERFIELD"
```

```
120 Q$=CHR$(34):S$="      ":B$(1)=Q$:
    ¬B$(3)=CHR$(58)
130 INPUT"VARIABLES OR LINES";Z$:C2=5:
    ¬IFASC(Z$)=76THENC2=6
140 FORJ=1TO255:C(J)=4:NEXTJ:FORJ=48TO57:
    ¬C(J)=6:NEXTJ
150 IFC2=5THENFORJ=65TO90:C(J)=5:NEXTJ:
    ¬FORJ=36TO38:C(J)=7:NEXTJ:C(40)=8
160 C(34)=1:C(143)=2:C(131)=3
170 INPUT"PROGRAM NAME";P$:OPEN1,8,3,"0:
    ¬"+P$+",P,R"
180 GET#1,A$,B$:IFASC(B$)<>4THENCLOSE1:
    ¬STOP
190 IFB=0GOTO240
200 PRINTL$;:K=X:FORJ=BTO1STEP-1:PRINT" ";
    ¬A$(J);:X$=A$(J)+L$
210 IFX$(K)>=X$THENX$(K+J)=X$(K):K=K-1:
    ¬GOTO210
220 X$(K+J)=X$:NEXTJ:X=X+B:PRINT:B=0
230 REM: GET NEXT LINE, TEST END
240 GET#1,A$,B$:IFLEN(A$)+LEN(B$)=0GOTO530
250 REM GET LINE NUMBER
260 GET#1,A$:L=LEN(A$):IFL=1THENL=ASC(A$)
270 GET#1,A$:A=LEN(A$):IFA=1THENA=ASC(A$)
280 C=C2:C1=-1:L=A*256+L:L$=STR$(L):
    ¬IFLEN(L$)<6THENL$=LEFT$(S$,6-LEN(L$)
    ¬)+L$
290 REM GET BASIC STUFF
300 GET#1,A$:A=LEN(A$):IFA=1THENA=ASC(A$)
310 C9=C(A):IFC9>C1GOTO380
320 K=0:IFB=0GOTO360
330 FORJ=1TOB:IFA$(J)=M$GOTO370
340 IFA$(J)<M$THENNEXTJ:K=B:GOTO360
350 FORK=BTOJSTEP-1:A$(K+1)=A$(K):NEXTK
360 B=B+1:A$(K+1)=M$
370 C=C2:C1=-1:M$=""
380 IFC2=5GOTO420
390 IFA=137ORA=138ORA=141ORA=167THENC=6:
    ¬GOTO470
400 IFA=44ORA=32GOTO470
410 IFC9<>6THENC=9:GOTO470
420 IFC9=CTHENC=-1:C1=4
430 IFC>6GOTO470
440 IFC<0ANDC9>C1ANDC9>6THENC1=C9:GOTO460
450 IFC2=5THENIFLEN(M$)>2ORC>0GOTO470
460 M$=M$+A$
470 ONC9+1GOTO190,480,480,480:GOTO300
```

```
480 B$=B$(C9):C$=""
490 GET#1,A$:IFA$=""GOTO190
500 IFA$=B$GOTO300
510 IFA$=<>Q$GOTO490
520 A$=B$:B$=C$:C$=A$:GOTO490
530 CLOSE1:INPUT"PRINTER";Z$
540 C=3:Z=6:IFASC(Z$)=89THENC=4:Z=12
550 OPEN2,C:PRINT#2:PRINT#2,"CROSS ¬
    ¬REFERENCE - PROGRAM ";P$
560 X$="":FORJ=1TOX:A$=X$(J)
570 FORK=1TOLEN(A$):IFMID$(A$,K,1)<>" ¬
    ¬"THENNEXTK:STOP
580 B$=LEFT$(A$,K-1):C$=MID$(A$,K+1):
    ¬IFX$=B$GOTO600
590 PRINT#2:Y=0:X$=B$:PRINT#2,X$;LEFT$(S$
    ¬5-LEN(X$));
600 Y=Y+1:IFY<ZGOTO620
610 Y=1:PRINT#2:PRINT#2,S$;
620 PRINT#2,LEFT$(S$,6-LEN(C$));C$;
630 NEXTJ:PRINT#2:CLOSE2
```



Trace For The PET

Brett Butler

I wished I had a TRACE program when I first got my PET. Eventually, I wrote it myself.

TRACE allows you to actually see BASIC executing. It resides in the high end of memory, occupying less than 340 bytes.

It displays each line as it's interpreted. This means that it shows the actual BASIC commands being performed, rather than just LISTing the line. If part of a line is not executed, you won't see it. For example, if you have a conditional statement such as:

100 ON A GOTO 200,300,400 and variable
A is 2, you'll see: 100 ON A GOTO 200,300, or with an IF statement like:
100 IF A > 5 THEN B = B + 2
with A less than 5 you'll see: 100 IF A > 5 THEN B
with A 5 or over you'll see: 100 IF A > 5 THEN B = B + 2

One more characteristic of TRACE: it also shows values that are being input.

TRACE comes as a BASIC program, which POKEs the machine language instructions into the proper place. It finds the high end of memory, wherever it happens to be, and then builds the machine language up there. So it doesn't matter if your PET is fitted with 4K, 8K, 16K or more: TRACE will be packed into the right place.

Programs may be changed, or new programs loaded, without affecting TRACE. It will stay in there until you cut power off. All BASIC functions operate normally (but slower). If you use the STOP key to stop a program, hold it down for a few moments until it "catches."

There are two versions of TRACE: one for original ROM and one for the new upgrade (16K, 32K) ROM.

Once the machine language version of TRACE is written to fit your machine, it may be used right away or saved with the Machine Language Monitor . . . BASIC TRACE tells you how to do this. The machine language version is handier, since it will load more quickly — and it may be loaded without disturbing other BASIC programs previously in memory.

There are four locations you need to know to run TRACE properly. The BASIC TRACE loader gives you the addresses that apply to your machine.

INITIALIZE — seals TRACE into high memory and restores any existing BASIC programs. Use once after loading the machine

language TRACE.

ARM — sets TRACE on. From this point on, BASIC programs can be TRACE'd.

DISARM — sets TRACE off. TRACE remains locked in high memory, but does not act on your BASIC program.

Speed Location — Poke any value from 1 to 255 here, to control the speed of the TRACE display.

The SYS commands for ARM and DISARM may be given directly from a program. So when you're debugging, you can have your program turn TRACE on at a certain point, and turn it off again later.

If you're tracing a dull part of your program, hold down the SHIFT key. This will speed things up a bit.

Special thanks to Jim Butterfield, Toronto; without his encouragement and assistance TRACE would still be just an idea.

```
1 PRINT"THIS PROGRAM LOCATES TRACE IN"
2 PRINT"ANY SIZE MEMORY THAT IS FITTED..."
3 PRINT"THIS VERSION WORKS ONLY WITH"
4 PRINT"ORIGINAL R Ö M PETS - USE ANOTHER"
5 PRINT"VERSION FOR THE NEW (16K,32K) MACHINES"
10 DATA-343,162,5,189,181,224,149, 194,202,16,248,169,239,133,210,96
11 DATA173,-343,133,134,173,-342,133, 135,169,255,133,124,160,0,162
12 DATA3,134,125,162,3,32,-272,208, 249,202,208,248,32,-272,32,-272
13 DATA76,106,197,162,5,189,-6,149, 194,202,16,248,169,242,133,210,96
14 DATA230,124,208,2,230,125,177,124, 96,230,201,208,2,230,202,96,32
15 DATA197,0,8,72,133,79,138,72,152, 72,166,137,165,136,197,77,208,4
16 DATA228,78,240,107,133,77,133,82, 134,78,134,83,173,4,2,208,14,169
17 DATA3,133,74,202,208,253,136,208, 250,198,74,16,246,32,-54,169,160
18 DATA160,80,153,255,127,136,208, 250,132,76,132,84,132,85,132,86,120
19 DATA248,160,15,6,82,38,83,162, 253,181,87,117,87,149,87,232,48,247
20 DATA136,16,238,216,88,162,2,169, 48,133,89,134,88,181,84,72,74,74
21 DATA74,74,32,-44,104,41,15,32, -44,166,88,202,16,233,32,-38,32,-38
22 DATA165,75,197,201,240,55,165, 79,208,4,133,77,240,47,16,42,201,255
23 DATA208,8,169,94,32,-30,24,144, 33,41,127,170,160,0,185,145,192,48
24 DATA3,200,208,248,200,202,16, 244,185,145,192,48,6,32,-32,200,208
25 DATA245,41,127,32,-32,165,201, 133,75,104,168,104,170,104,40,96,168
26 DATA173,64,232,41,32,208,249,152, 96,9,48,197,89,208,4,169,32,208
27 DATA2,198,89,41,63,9,128,132,81, 32,-54,164,76,153,0,128,192,79,208
28 DATA2,160,7,200,132,76,164,81,96, 76,-256,32,-263
1000 S2=PEEK(134)+PEEK(135)*256: S1=S2-343
1010 FOR J=S1 TO S2-1
1020 READ X:IF X>=0 GOTO 1050
1030 Y=X+S2:X=INT(Y/256):Z=Y-X*256
1040 POKE J,Z:J=J+1
1050 POKE J,X
1060 NEXT J
1070 PRINT" === TRACE ==="
1080 REM BY BRETT BUTLER, TORONTO
1090 PRINT"TO INITIALIZE AFTER LOAD: SYS";S1+17
1100 PRINT"TO ENABLE TRACE:      SYS";S1+56
1110 PRINT"TO DISABLE:     SYS"; S1+2
1120 PRINT"CHANGE SPEED WITH: POKE"; S1+124;",X"
1130 PRINT"==MAKE A NOTE OF ABOVE COMMANDS=="
1140 PRINT"SAVE USING MACHINE LANGUAGE MONITOR:"
1150 PRINT"   .S 01,TRACE";
1160 S=INT(S1/256):T=S1-S*256
```

```
1170 POKE 134,T:POKE 135,S
1180 POKE 130,T:POKE 131,S
1190 S=S1:GOSUB1400
1200 S=S2:GOSUB1400
1210 PRINT:END
1400 PRINT",";:S=S/4096
1410 GOSUB1420
1420 GOSUB1430
1430 T=INT(S):IF T>9 THEN T=T+7
1440 PRINT CHR$(T+48);: S=(S-INT(S))*16:RETURN
```

```
5 PRINT"THIS PROGRAM LOCATES TRACE IN"
6 PRINT"ANY SIZE MEMORY THAT IS FITTED..."
7 PRINT"THIS VERSION WORKS ONLY WITH"
8 PRINT"UPGRADE R O M (32K) PETS - USE
9 PRINT"ANOTHER VERSION FOR ORIGINAL R O M"
10 PRINT"MACHINES."
11 DATA -342,162,5,189,249,224,149, 112,202,16,248,169,239,133,128,96
12 DATA 173,-342,133,52,173,-341,133, 53,169,255,133,42,160,0,162,3
13 DATA 134,43,162,3,32,-271,208,249, 202,208,248,32,-271,32,-271,76
14 DATA 121,197,162,5,189,-6,149,112, 202,16,248,169,242,133,128,96
15 DATA 230,42,208,2,230,43,177,42, 96,230,119,208,2,230,120,96
16 DATA 32,115,0,8,72,133,195,138,72, 152,72,166,55,165,54,197
17 DATA 253,208,4,228,254,240,106, 133,253,133,35,134,254,134,36,165
18 DATA 152,208,14,169,3,133,107,202, 208,253,136,208,250,198,107,208
19 DATA 246,32,-54,169,160,160,80, 153,255,127,136,208,250,132,182,132
20 DATA 37,132,38,132,39,120,248,160, 15,6,35,38,36,162,253,181
21 DATA 40,117,40,149,40,232,48,247, 136,16,238,216,88,162,2,169
22 DATA 48,133,103,134,102,181,37,72, 74,74,74,74,32,-44,104,41
23 DATA 15,32,-44,166,102,202,16,233, 32,-38,32,-38,165,184,197,119
24 DATA 240,55,165,195,208,4,133,253, 240,47,16,42,201,255,208,8
25 DATA 169,105,32,-30,24,144,33,41, 127,170,160,0,185,145,192,48
26 DATA 3,200,208,248,200,202,16,244, 185,145,192,48,6,32,-32,200
27 DATA 208,245,41,127,32,-32,165, 119,133,184,104,168,104,170,104,40
28 DATA 96,168,173,64,232,41,32,208, 249,152,96,9,48,197,103,208
29 DATA 4,169,32,208,2,198,103,41,63, 9,128,132,106,32,-54,164,182
30 DATA 153,0,128,192,195,208,2,160, 7,200,132,182,164,106,96,76
31 DATA -255,32,-262
1000 S2=PEEK(52)+PEEK(53)*256: S1=S2-342
1010 FOR J=S1 TO S2-1
1020 READ X:IF X>=0 GOTO 1050
1030 Y=X+S2:X=INT(Y/256):Z=Y-X*256
1040 POKE J,Z:J=J+1
1050 POKE J,X
1060 NEXT J
1070 PRINT" === TRACE ==="
1080 REMARK: BY BRETT BUTLER, TORONTO
1090 PRINT"TO INITIALIZE AFTER LOAD: SYS";S1+17
1100 PRINT"TO ENABLE TRACE:      SYS";S1+56
1110 PRINT"TO DISABLE:     SYS";S1+2
1120 PRINT"CHANGE SPEED WITH:  POKE";S1+123;",X"
1130 PRINT"==MAKE A NOTE OF ABOVE COMMANDS=="
1140 PRINT"SAVE USING MACHINE LANGUAGE MONITOR:"
1150 PRINT"   .S ";
1160 S=INT(S1/256):T=S1-S*256
1170 POKE 52,T:POKE 53,S
1180 POKE 48,T:POKE 49,S
1190 PRINTCHR$(34);"TRACE";CHR$(34); ",01";
1200 S=S1:GOSUB1400
1210 S=S2:GOSUB1400
1220 PRINT:END
1400 PRINT",";:S=S/4096
1410 GOSUB1420
1420 GOSUB1430
1430 T=INT(S):IF T>9 THEN T=T+7
1440 PRINTCHR$(T+48);:S=(S-INT(S))*16 :RETURN
```

Utinsel: Enabling Utilities

Larry Isaacs

There is a growing amount of good utility software which can make the time spent with our PETs more productive. Some of these utilities add themselves to the operating system of the PET, providing us with extra commands, debugging tools, etc. It was may desire to have a number of these in memory at the same time and be able to access them as needed. To do this, there were a couple of problems to be dealt with.

The first problem comes from the way these utilities attach themselves to the operating system. When enabled, usually via a SYS command, they attach themselves to the operating system by modifying the CHARGOT routine found in page zero. The operating system uses this routine to fetch characters from a program while it is running, or from the keyboard buffer when executing immediate commands. By modifying the CHARGOT routine, the utility can examine the input before the operating system does. When using various utilities, it's possible for one utility's modifications to be incompatible with another's.

So far the only difficulty I've encountered involving the CHARGOT routine is with Commodore's DOS Support Program, also known as the Wedge. The Wedge requires a machine language jump instruction in the first three bytes of the CHARGOT routine. This jump instruction should jump to the starting point in the Wedge machine code. Unfortunately, the Wedge is not able to put this jump instruction into the CHARGOT routine. It is put there by some extra machine code which is part of the DOS Support Program on diskette. This means I don't have a SYS command to enable the Wedge. If I enable another utility which modifies the first three bytes of the CHARGOT routine, it would take some extra work to re-enable the Wedge.

The second problem is simply all those SYS commands you have to remember. The following program, called UTINSEL, provides a simple and flexible solution to the above problems. With it, you only need to remember a couple of SYS commands.

UTINSEL consists of a menu table and a machine language program, which is executed via a SYS command. The menu table is

user definable and can contain up to nine entries. Each entry consists of a prompt mesesage plus a copy of the CHARGOT routine that enables the associated utility or utilities. When executed, the machine language program lists the prompt messages, preceding each with a number. By typing the number of the menu item you want plus a carriage return, the associated CHARGOT routine will be copied into the proper location in page zero.

The UTINSEL/NEW program includes the machine code and menu table in DATA statements, plus a BASIC program which POKES the code and table into the top of free memory. To adapt UTINSEL/NEW to your own requirements, you need only modify the menu table.

To set up your own utility package, first reset your PET. Next, load in the utilities which occupy RAM. Now run a version of UTINSEL with only the ORIGINAL menu entry provided in the listing. To determine what CHARGOT routines you will need for your menu table, write a short program to print out 24 memory locations starting at 112 for upgrade PETs and 194 for original PETs. Now you may enable the desired utility and then run your program to print out the required CHARGOT routine. Before enabling other utilities, restore the original CHARGOT routine by executing the UTINSEL program you loaded earlier. In some cases it is possible to enable more than one utility. For example, The BASIC Programmer's Toolkit and the WEDGE can be enabled simultaneously.

To set up the menu table, you must include a set of DATA statements for each utility. The first DATA statement of a set should contain a prompt message. The next DATA statements should contain the CHARGOT routine needed by the utility, or utilities, associated with the prompt message. These sets of statements may be placed in the menu table in any order. After you've entered the menu table, set N in the program to the number of entries.

Now save a copy of the program with your table; then run it. The machine code and table will be loaded into memory just below your utilities. Write down the two SYS commands it prints out. The first one sets the top of memory pointer to just below UTINSEL, and the other executes the UTINSEL machine code. To save your utility package, enter the machine language monitor and examine the top of memory pointer at hex 34 and 35. If you haven't done this before, type SYS1024 and M 0034 0035 to get a hex dump of these locations. Use these values to save memory from this starting address up to the physical top of memory. Refer to the Commodore manual for more

detail on the SAVE command. For the program listing provided, hex 34 and 35 were 47 and 7D respectively, and the save command was S "UTILITY.PKG",08,7d47,8000 for saving on Commodore disk using a 32K CBM. Whenever you load in your utility package, be sure to use the SYS command to get the top of free memory before running any programs.

The listing provided is for UTINSEL/NEW, which runs on new ROMs. The menu table includes entries for The BASIC Programmer's Toolkit, the Commodore DOS Support program, and the original CHARGOT routine. You may also want to include the TRACE utility by Brett Butler in the Fall COMPUTE! The second listing gives the changes needed to convert the program to UTINSEL/OLD for original ROMs. When printing the CHARGOT routine, start at 194 instead of 112. You will have to have a machine language monitor on tape if you wish to save the machine code on the original ROMs.

```
100 REM UTINSEL/NEW
110 REM UTILITY INPUT ROUTINE SELECTOR
120 PRINT
130 PRINT"COPYRIGHT 1979 SMALL SYSTEM"
140 PRINT"SERVICES,INC."
150 PRINT"900 SPRING GARDEN STREET"
160 PRINT"GREENSBORO, N.C. 27403 USA"
170 PRINT
180 REM ALL RIGHTS RESERVED. THIS
190 REM PROGRAM MAY BE DUPLICATED FOR
200 REM USE BY INDIVIDUALS FOR THEIR
210 REM SPECIFIC MACHINE. SUCH
220 REM DUPLICATION MUST INCLUDE THE
230 REM COPYRIGHT NOTICE AND ADDRESS.
240 REM REPRODUCTION FOR COMMERCIAL
250 REM PURPOSES IS EXPRESSLY
260 REM PROHIBITED.
270 REM
280 PRINT"UTINSEL IS BEING LOADED"
290 PRINT"INTO HIGH MEMORY"
300 PRINT
310 REM
320 REM UTINSEL MACHINE CODE
330 DATA 169,0,133,48,133,52,169,0
340 DATA 133,49,133,53,76,137,195,165
350 DATA 1,72,165,2,72,162,0,160
360 DATA 0,134,1,132,2,162,1,142
370 DATA 58,3,169,13,32,210,255,32
380 DATA 210,255,32,-38,224,0,208
```

```
390 DATA 23,169,13,32,210,255,169,145
400 DATA 32,210,255,173,58,3,9,48
410 DATA 32,210,255,238,58,3,208,218
420 DATA 169,13,32,210,255,32,207,255
430 DATA 170,41,240,201,48,208,241,138
440 DATA 41,15,205,58,3,16,233,141
450 DATA 58,3,32,-38,162,0,177
460 DATA 1,149,112,224,24,240,4,232
470 DATA 200,208,244,104,133,2,104,133
480 DATA 1,76,137,195,160,0,174,58
490 DATA 3,202,240,9,177,1,201,0
500 DATA 240,22,168,208,244,32,205,253
510 DATA 32,205,253,200,177,1,201,0
520 DATA 240,6,32,210,255,24,144,243
530 DATA 200,96
540 REM
550 REM MENU TABLE
560 DATA "TOOLKIT"
570 DATA 230,119,208,2,230,120,173,0
580 DATA 0,76,154,178,0,76,196,178
590 DATA 100,0,100,0,56,233,179,0
600 DATA "WEDGE 4.0":REM FOR 32K PET
610 DATA 76,82,126,2,230,120,173,0
620 DATA 0,201,58,176,10,201,32,240
630 DATA 239,56,233,48,56,233,208,96
640 DATA "ORIGINAL"
650 DATA 230,119,208,2,230,120,173,0
660 DATA 0,201,58,176,10,201,32,240
670 DATA 239,56,233,48,56,233,208,96
680 REM
690 REM POKE MACHINE CODE TO TOP OF
700 REM FREE MEMORY
710 REM
720 TA=PEEK(52)+PEEK(53)*256-1
730 SA=TA-162
740 FORJ=SATOTA-1
750 READ B:IFB>=0 GOTO790
760 AD=B+TA: B=INT(AD/256)
770 B1=AD-B*256
780  POKE J,B1:J=J+1
790 POKE J,B
800 NEXT J
810 REM
820 REM LOAD TABLE FROM TOP DOWN
830 REM FIRST MOVE POINTER FOR STRINGS
840 REM
850 T1=INT((SA-2048)/256)
```

```
860 T2=(SA-2048)-T1*256
870 POKE48,T2:POKE49,T1
880 REM
890 REM SET N = #TABLE ENTRIES
900 REM
910 N=3
920 S1=SA
930 FORK=1TO N
940 READ M$:EL=LEN(M$)+26
950 S2=S1:S1=S2-EL
960 POKES1,EL:IF K=1 THEN POKES1,0
970 FORJ=1TOLEN(M$)
980 POKES1+J,ASC(MID$(M$,J))
990 NEXT J
1000 S3=S1+J:POKES3,0:S3=S3+1
1010 FOR J=S3TOS2-1
1020 READ B:POKEJ,B
1030 NEXT J
1040 NEXT K
1050 REM
1060 REM FIX POSITION DEPENDENT CODE
1070 REM
1080 T1=INT(S1/256):T2=S1-T1*256
1090 POKE SA+22,T2:POKE SA+24,T1
1100 POKE SA+1,T2-1:POKE SA+7,T1
1110 REM
1120 REM LINK TABLE
1130 REM
1140 LA=S1:L=0
1150 IF PEEK(LA)=0 THEN GOTO 1190
1160 L=L+PEEK(LA):POKE LA,L:LA=S1+L
1170 GOTO1150
1180 REM
1190 PRINT "USE SYS" SA;
1200 PRINT "TO SET TOP OF MEMORY"
1210 PRINT"USE SYS" SA+15;
1220 PRINT "TO RUN UTINSEL"
1230 POKE52,T2-1:POKE53,T1
1240 POKE48,T2-1:POKE49,T1
1250 NEW
```

```
1 REM UTINSEL NEW-TO-OLD
2 REM COPYRIGHT 1979 SMALL SYSTEM
3 REM SERVICES,INC.
4 REM 900 SPRING GARDEN STREET
5 REM GREENSBORO, N.C. 27403 USA
7 REM TO CONVERT UTINSEL/NEW TO
8 REM UTINSEL/OLD, SUBSTITUTE
```

```
9 REM THE FOLLOWING STATEMENTS
330 DATA 169,0,133,130,133,134,169,0
340 DATA 133,131,133,135,76,139,195,165
380 DATA 210,255,32,-40,224,0,208
450 DATA 58,3,32,-40,162,0,177
480 DATA 1,76,139,195,160,0,174,58
500 DATA 240,24,168,208,244,169,32,32
510 DATA 210,255,32,210,255,200,177,1
520 DATA 201,0,240,6,32,210,255,24
530 DATA 144,243,200,96
570 DATA 230,201,208,2,230,202,173,0
600 REM DISCARD WEDGE MENU ENTRY
650 DATA 230,201,208,2,230,202,173,0
720 TA=PEEK(134)+PEEK(135)*256-1
730 SA=TA-164
870 POKE130,T2:POKE131,T1
1230 POKE134,T2-1:POKE135,T1
1240 POKE130,T2-1:POKE 131,T1
```

Converting ASCII Files to PET BASIC

Harvey B. Herman

Recently I have been experimenting with a program (not discussed here) which makes PET into a terminal (CompuMart T/C 2001 terminal option) which can communicate with remote computers. Normally, the characters that are received by the PET, when acting as a terminal, are displayed on the screen. I modified the program to optionally save the characters (ASCII Code) to a reserved area in high memory (approximately decimal 8192 and above). Obviously, this program required memory in this area and will need to be modified for an unexpanded 8K PET. The question one might ask is, "What can I do with an ASCII file in high memory?" This article is intended to answer that question.

Commodore's PET is not the first computer I have worked with and I suspect the same may be true for many readers. I have spent many hours developing BASIC programs on remote computers for use with my research and in my teaching. It would be advantageous if I could also use these programs (suitably modified) on the PET. I have no strong desire to retype all these programs. If I could convert the ASCII file of a program listing made by a terminal program into a PET BASIC program it would save immense amounts of work. Any minor changes could then be done with the screen editor.

The program called ASCII shown in the figure, converts ASCII files in high memory into BASIC programs. It is intended for use with expanded PETs with original ROMs. The POKE locations in statement number 63290, 525-527-528, need to be changed to 158-623-624 for upgrade ROMs. The program uses the dynamic keyboard idea of Mike Lauder (see Best of PET Gazette). It writes two lines on the screen and puts two carriage returns in the keystroke buffer. The first line is a BASIC statement taken from part of a program listing saved in high memory by the terminal program. The second line is an immediate mode statement which restores a memory position counter and jumps back into the main program again. It is necessary to remember the current position in high memory because all variables were set to zero after the previous step. This is true whenever a new BASIC statement is added to a program, as in this case. All the new BASIC statements are added

to the front of the main program which was purposely written with very large statement numbers. At the conclusion of this program, the statements belonging to the ASCII program can be deleted by hand or with The Programmer's BASIC Toolkit.

The ASCII program can be used to do a minor amount of editing "on the fly." Some of my original programs were done on a computer which uses "#" instead of "<>" and I included a conversion in statement 63180. Also "[" and "]" were used in place of "(" and ")," in some places and this conversion is done in statements 63160 and 63170. I also removed 7 (bell character) from the programs. Besides giving a syntax error later when run on a PET program, the inclusion of 7 occasionally caused lines to be over 80 characters long. This stopped the ASCII program with a syntax error which then had to be manually restarted. All these programmed editing changes saved a lot of manual editing later.

The end of my ASCII files is signified by the ASCII character 4, otherwise the program might continue indefinitely, adding unwanted BASIC statements, or garbage. This check is done in statement 63200. It should reach this point and stop if each line begins with a number, is less than 80 characters long, and the counter in statement 63070 is positioned to the beginning of the ASCII listing in high memory. Conversion to PET syntax, if required, would begin here.

I have used the ASCII program to convert very large ASCII files to PET programs. The same program should be useful when I acquire CP/M ASCII files on 5-1/4" diskettes. The disk and operating system which I am using (PEDISK and Wilserv Software) can read CP/M files, and the ASCII program discussed here will convert them to PET BASIC programs.

```
63000 REM PROGRAM CONVERTS AN ASCII FILE ¬
      ¬IN
63010 REM HIGH MEMORY TO A PET BASIC ¬
      ¬PROGRAM
63020 REM HIGH MEMORY BEGINS AT $2017(8215
      ¬ DEC)
63030 REM
63040 REM HARVEY B. HERMAN
63050 REM
63060 REM I IS MEMORY COUNTER
63070 I=8215
63080 REM THROW AWAY FIRST LINE
63090 A=PEEK(I):IFA<>13THENI=I+1:GOTO63090
63100 PRINT"ñ↓↓↓"
63110 I=I+1
```

```
63120 REM NEXT CHARACTER FROM ON HIGH
63130 A=PEEK(I)
63140 REM REPLACE [ & ] WITH ( & )
63150 REM REPLACE # WITH <>
63160 IF A=91 THEN A=40
63170 IF A=93 THEN A=41
63180 IF A=35 THEN PRINT"<>";:GOTO63110
63190 REM CHAR $04 AT END OF FILE
63200 IF A=4 THEN STOP
63210 REM THROW AWAY '7
63220 IF A=39 THEN IF PEEK(I+1)=55 THEN ¬
      ¬I=I+1:GOTO 63110
63230 REM PRINT BASIC LINE ON SCREEN. ¬
      ¬AFTER CR
63240 REM PRINT NECESSARY VARIABLES AND ¬
      ¬PUT CR
63250 REM IN KEYSTROKE BUFFER. END PROGRAM
63260 REM INCORPORATE LINE AND BEGIN AGAIN
63270 PRINT  CHR$(A);
63280 IF A=13 THEN PRINT"I=";I;":GOTO63100
      ¬h↓"
63290 IF A=13 THEN POKE 527,13:POKE528,13:
      ¬POKE 525,2:END
63300 GOTO63110
```

Multitasking On Your PET? QUADRA-PET

Charles Brannon

QUADRA-PET is a machine language program that lets you partition the memory of an Upgrade ROM PET or CBM into four 8K blocks. Each block is an independent program workspace. Programs existing in each 8K partition can be selected and then used and modified without affecting any of the other programs. You can jump to any other of the programs at any time.

After initialization with SYS 926, PET displays the question:

WHICH PET? [1-4]

Perhaps Mary, an avid computer-games buff, types in "1" and loads STARTREK. She plays it for a while and then leaves to eat lunch. Meanwhile, Bob goes to the PET, sees that someone is using PET #1, and switches to PET #2 to write a business program. After nearly "perfecting" it, he leaves to see what Mary is up to. Now the kids come in and, after arguing for a half-hour, agree to share the PET, one using PET #3 and the other PET #4. Fortunately for Bob and Mary, nothing the kids do can harm their programs.

How To Use QUADRA-PET

1. Load or type in one of the versions of QUADRA-PET (BASIC or hex):
2. Enter NEW
3. SYS 926 to initialize.
4. PET will respond with WHICH PET? (1-4)
5. Select the one you wish to use.
6. Before loading or typing in a program for the first time, type in NEW.
7. To select another PET, SYS 826 and follow instructions 4-7.

Now comes the fun part — how does it work? Many memory locations in zero page (0-256) are *pointers*. QUADRA-PET works with three of those pointers.

On power-up, PET determines the end of memory by writing a character to every memory location and then reading it back. PET

then increments a memory location until a failure in reading that character occurs. This indicates that the end of available memory has been reached. Physically, this pointer is at location 52 decimal ($34). The second pointer is at the start of memory, stored in location 41. Originally, this points to the actual start of user memory, 1024. The last pointer is the end of text pointer. As you write your program it changes.

QUADRA-PET partitions the memory by changing these pointers to point to successively higher memory locations, depending on which PET is in use. Since the end of text pointer changes, it must be saved before we move to a new PET and restored on return. QUADRA-PET, as it is in machine language, does all these things seemingly instantaneously.

HOW TO SAVE A PROGRAM PRODUCED WITH QUADRA-PET:

1. SYS 1024 to go to the Monitor.
2. Enter: .M 0028 002B and type RETURN.
3. You will get a display something like:
   .:0028 01 04 3E 04 .. ..
4. We will use only the first four bytes. Write down the first pair in reverse order on paper, for example:
   0401
   Do the same with the second pair. (e.g. 043E)
5. Enter: .S "PROG NAME",01,XXXX,YYYY where "PROG NAME" is the name of your program, XXXX is the first number you wrote down, and YYYY is the second. For example, to save the example program which we will call "PET#1," you would enter: .S "PET #1", 01,0401,043E
6. Press RETURN and press play and record to save your program.
7. To load *this* saved program into a space prepared by QUADRA-PET, just SYS 1024 and enter .L "PROG NAME" where "PROG NAME" is the name of your program.

HOW TO LOAD A PRE-EXISTING PROGRAM INTO A SPACE PREPARED BY QUADRA-PET:

I could tell you how to do this on the original ROM PET but, quite frankly, I can't find the memory locations for this procedure in the new PET. All you PET experts — HELP!

If you can figure it out, please send in the procedure to **COMPUTE!.**

A little imagination will create many uses for QUADRA-PET.

For education, it is the perfect way to keep four students' programs in the PET at the same time. Each program can be worked on and modified in any way without affecting any of the other

programs.

In business, four different business programs can exist simultaneously in PET's memory, ready to use. For the small penalty of loading the programs into the program workspaces at the start of the day, all four are within reach of a carriage return — faster than any disk drive.

Machine language programmers can fill partitions with useful routines, leaving one or more partitions for BASIC. QUADRA-PET itself is short and easily relocatable.

I would be interested to find out which novel and useful applications for QUADRA-PET *you* can think up!

Happy QUADRA-PETing!

References

CBM User Manual 2001-32, First Edition. Commodore Business Machines, Inc., Palo Alto, CA (1979)

Harvey B. Herman, "Memory Partition of BASIC Workspace", **COMPUTE!**, pp. 18-20 (Jan., Feb. 1980)

Jim Butterfield, "PET in Transition (memory map) **COMPUTE!**, pp. 68-70 (Fall, 1979)

```
0 REM*********************************
1 REM            QUADRA PET
2 REM*********************************
3 REM:BY CHARLES BRANNON 06/07/80
10 FOR I =  826 TO 941
20 READ A
30 POKE I, A
40 NEXT
50 SYS926
60 END
1000 DATA174,126,3,165,42,157,131,3,165
1010 DATA43,157,135,3,169,143,160,3,32
1020 DATA28,202,32,228,255,41,15,240,249
1030 DATA201,5,176,245,170,202,142,126,3
1040 DATA169,1,133,40,189,127,3,133,41
1050 DATA189,131,3,133,42,189,135,3,133
1060 DATA43,169,0,133,52,189,139,3,133
1070 DATA53,32,119,197,96,0,4,32,64
1080 DATA96,3,3,3,3,4,32,64,96
1090 DATA32,64,96,128,87,72,73,67,72
1100 DATA32,80,69,84,63,32,40,49,45
1110 DATA52,41,0,169,0,141,0,32,141
1120 DATA0,64,141,0,96,76,58,3
READY.
```

An Easier Method of Saving Data Plus Home Accounting

Robert W. Baker

The techniques presented here are applicable to a wide variety of systems where non-volatile variables are needed.

Whenever a program must save specific data for the next time it is run, the data are normally saved in a tape data file. This requires inserting a tape with the data file and reading the previous values every time the program is run. When the program is done, the tape must be rewound or changed and another data file written to save the new data. This procedure can waste a good deal of time, especially if only a small amount of data is needed and the program is normally run quite frequently. It can take as long as 10 to 15 seconds on the PET just to find the data file and read the header record before actually reading any data. In addition, you now have a tape for the program and another tape for the data file. If you only have several values to save, using data files is awkward, cumbersome, and not worth the trouble. Several applications that could be done very easily on a computer, in reality become useless when requiring data files.

Another possible method of saving data is to change the BASIC pointers and save the data along with the program on tape. The next time the program is loaded, the BASIC pointers are then reset and a GOTO xxxxx command is used to execute the program. If a RUN command is used instead of a GOTO, the data is lost and the program must be re-loaded. This method is too complicated and requires a number of functions the user must perform each time he or she saves or loads the program and data. It also runs the risk of permanently losing the data.

There is a way, however, that a program can save small amounts of data within the program itself using a very simple procedure. The basic theory is to include DATA statements in the program with initial data specified for the first time the program is run. The DATA statements and their associated data define space within the program for the data that is to be saved after each time

the program is run. Before terminating, the program simply POKE's the new values to be saved back into the DATA statement(s) to replace the original data. The program itself is then saved after each execution and the latest data is automatically included without any special actions by the user. Whenever the program is loaded, the previous data is readily available using the standard READ command of BASIC. Saving data using this method is extremely simple, but it does require knowing the format of BASIC lines stored in memory. The necessary information on the PET has been described in various newsletters and several magazine articles, so it will not be repeated here. This article was written for the PET primarily, but the technique will work for other machines as well.

The listing for a Home Budget program that I've been using for several months on my 8K PET is included to help illustrate this simple data saving technique. Looking at the start of the program, lines 10 and 20 contain DATA statements to reserve space for 12 numeric values to be saved after each time the program is run. The DATA statements are located at the very beginning of the program, making it easier to know where to do the POKE's. Each value saved can be up to six digits in length, since this is the length of each field specified in the original DATA statements. Disregarding how the program actually works for now, the data from lines 10 and 20 would normally be read into elements of array "M" by lines 510 and 520. When the program is done, the value of a single element of M or M(x) is converted to a six character string with leading zeros blanked as spaces in line 1010. This insures that all six characters of each field in the data statements are changed every time the new data is saved in the program. The ASCII value of each character in the string representation of M(x) is then poked into a DATA statement by line 1030. This loop is repeated for all 12 values and a reminder is then printed so the user will not forget to save the program with the updated data. The program could have even printed the actual SAVE command for the user if desired. I intentionally left this out, however, in case the user decided the new data was not correct and wanted to re-run the program without saving the updated values.

If you should use this technique in your own program, don't forget it can be used to save strings or numbers. Be careful you don't destroy the DATA statement itself or the separating commas when POKEing characters into a DATA statement. Also, don't forget to step over the end-of-line flag, the 2-byte link, the 2-byte line number, and the 1-byte DATA statement "token" when more than

one line is used to save data in the program. Each field definition should reserve enough space for the maximum length expected to be encountered by the program. Numeric values must be converted to strings before being saved. Quotes should be used at the beginning and end of each field when saving text strings. Don't forget to step past the quotes when POKEing the strings into the DATA statements. Strings should be changed to the length of the field being POKEd into by appending spaces as was done in the example with the numeric values after converting them to text strings. This will insure the entire field is updated each time the program is run and the correct data is always saved. The DATA statements must remain at a constant location in memory. Being at the beginning of the program avoids problems with changing locations caused by editing the program before the DATA statements. If the DATA statements are moved, the address used for the POKEs must be changed accordingly.

Home Accounting

I don't claim to be an accounting expert but the Home Budget program works and serves a very useful purpose for me. It is based on an original budget system I devised that used an accounting book to record all income and expenditures. Various "accounts" within the budget help allocate what money from each paycheck is to be reserved for which bills in order to meet the projected expenses. Accounts for bills that are paid at least once a month are kept in the family checking account where they are readily available. Accounts for all other bills, paid at longer intervals, are normally kept in the savings account until needed. An account is established for each major expenditure, such as: insurances, home mortgage, utilities, telephone, auto loan, auto expenses, charge accounts, Christmas presents, vacation, etc. All smaller expenses are grouped into a miscellaneous account that is kept in the checking account. An additional account is reserved in the savings account for all "excess" funds, as the true "savings" total.

This simple BASIC program provides all the desired functions to keep an accurate home budget with a minimum of effort. It does not have any fancy features, instead it provides the necessary information in an easy-to-use format. It displays each account total along with the current checking and savings balances for fast and easy verification. Each transaction is entered by selecting the appropriate account number and the value to credit (+) or debit (–) the specified account. Positive values indicate deposits (credit) and

negative values indicate expenses or bills paid (debit). The actual transactions are not recorded, only the running totals for each account are retained to keep the amount of saved data at a minimum. An additional feature of this program is the ability to set the amount to be credited to each account for a paycheck deposit. Thus, come payday, you simply enter the amounts deposited to the checking and savings account and the program does the rest. An account total can become negative if expenses exceed current funds allocated for that expense. This effectively indicates "borrowing" money from other accounts and should be corrected by transferring money from another account or changing the pay deposit value for the account. A negative checking or savings balance should be avoided as this indicates a very serious problem such as an overdrawn checking account. The first step in setting up the budget is to decide what accounts are needed and how many will be in checking or savings. In line 500 of the program, the variable "C" is defined as the number of budget accounts in checking (7), and "S" is defined as the number of accounts in savings (5). The variable "A" is computed as the total number of budget accounts (C + S = 12), and the money (M) and name (N$) arrays are dimensioned in the same line.

Since we are going to save the data within the program, we must define storage for the values in DATA statements. Line 10 contains the initial values for the checking accounts and line 20 is for the savings accounts. Separate DATA statements were used for checking and savings to allow easy addition or deletion of accounts as required. All values will be kept as whole numbers by multiplying each value by 100. This will help avoid decimal points and problems associated with fractions, besides making the data easier to save using POKEs. With six digits per field, the limiting values for any account value are: -999.99 to +9999.99 since the minus sign takes up one digit space for negative numbers.

The actual account names are stored in lines 100-210. Lines 100 and 110 are for the checking accounts while lines 200 and 210 define the savings account names. Each name should be limited to 28 characters for the program to function properly. In addition, the last checking account must be the MISC account and the last savings account must be the excess SAVINGS account. The amount to be deposited from a paycheck to each checking account is specified in line 300 with a zero value shown for the MISC account, the last value. This account automatically gets any remainder from the pay deposit after all the required checking

account deposits are made. If the pay deposit is not large enough to meet the required checking budget total, the difference is subtracted from the MISC account. Line 400 contains the corresponding savings pay deposit values, with a zero value for the excess SAVINGS account, the last value. This account acts just as the MISC account does for the checking account. Any savings pay deposit excess/shortage is added/subtracted to this account.

To customize the program for your own use, simply set the correct values of C and S in line 500. Then add or delete the required DATA fields in lines 10 and 20, and the account names in lines 100-210. Change any account names as required, but keep each to a maximum of 28 characters. Set the PAY deposit values for each account in lines 300 and 400 by taking into consideration the related expenses and frequency of payment. Remember to keep the MISC and SAVINGS accounts as the last accounts in the checking and savings, with zero pay deposit values for each. That should be all the changes required to convert the program for your own situation. Individual accounts can be added or deleted at any time by similar changes. Don't forget to set an account value to zero by transferring any money to other accounts before deleting the account. This will keep your checking and savings balances correct.

The program listing contains a number of REM lines to help document the program. If you should decide to use the example program, please don't bother entering these lines. They'll only make the program LOADing and SAVEing much longer, since the program will be about three times larger than needed. This is exactly what we tried to avoid by saving the data within the program to minimize tape usage. Once typed in, a few minutes experimenting with the program should clearly indicate how it works. Enter a few transactions, then type D and LIST lines 10 and 20 to see what was SAVEd within the program. If you have any problems, check for extra spaces in lines 10 and 20 or check the POKE address in line 1010.

```
10 DATA000000,000000,000000,000000,000000,
     000000,000000
20 DATA000000,000000,000000,000000,000000
30 :
31 REM ******************************
32 REM *    HOME BUDGET PROGRAM      *
33 REM *----------------------------*
34 REM *    BY: ROBERT W. BAKER      *
35 REM *        15 WINDSOR DRIVE     *
```

```
36 REM *         ATCO, NJ 08004        *
39 REM *********************************
50 :
51 REM ================================
52 REM DATA STATEMENTS TO SAVE VALUES
53 REM MUST BE THE FIRST STATEMENTS
54 REM IN THE PROGRAM TO MAKE THEM
55 REM EASY TO FIND.
56 REM ACCOUNT DESCRIPTIONS FOLLOW -
57 REM ================================
58 :
100 DATA "CHARGES"
102 DATA "GAS & AUTO EXPENSES"
105 DATA "MORTGAGE"
110 DATA "TELEPHONE", "UTILITIES"
115 DATA "AUTO LOAN","MISC"
200 DATA "AUTO INSURANCE"
205 DATA "HOMEOWNERS INSURANCE"
210 DATA "LIFE INSURANCE", "CHRISTMAS"
215 DATA "SAVINGS"
250 :
251 REM ================================
252 REM FOLLOWING DATA STATEMENT
253 REM CONTAINS STANDARD DEPOSIT
254 REM VALUES FOR PAY DEPOSIT.
255 REM ================================
256 :
300 DATA 25,40,150,10,50,45,0
400 DATA 30,7,25,15,0
450 :
451 REM ================================
452 REM MAJOR VARIABLE DEFINITIONS:
453 REM  C  = # OF ITEMS IN CHECKING
454 REM  S  = # OF ITEMS IN SAVINGS
455 REM  A  = TOTAL NUMBER OF 'ACCTS'
456 REM  CB = CHECKING BALANCE
457 REM  SD = TOTAL SAVINGS DEPOSIT
458 REM  M(.)  = CURRENT ACCT VALUES
459 REM  N$(.) = ACCT NAMES FROM DATA
470 REM ================================
471 REM READ VALUES FROM DATA
472 REM STATEMENTS TO INITIALIZE.
479 REM ================================
480 :
500 C=7:S=5:A=C+S:DIM M(A),N$(A)
505 CB=0:SD=0
510 FOR X=1 TO C:READ M(X):CB=CB+M(X)
515 NEXT
```

```
520 FOR X=C+1 TO A:READ M(X)
525 SD=SD+M(X):NEXT
540 FOR X=1 TO A:READ N$(X):NEXT
550 L$=".......................... $"
590 :
591 REM ============================
592 REM DISPLAY ACCT #, NAME, & VALUE
593 REM ALONG WITH CHECKING/SAVINGS
594 REM TOTALS, THEN PROMPT FOR INPUT
595 REM ============================
596 :
600 PRINT"[CLR]";:C2=0:S2=0
605 FOR X=1 TO C:V=M(X):GOSUB 900:NEXT
606 PRINT TAB(30)"["DDDDDDDDD"]"
610 PRINT"   [RV]TOTAL CHECKING ";
611 PRINT"BALANCE[RVOFF]..... $";
615 V=CB:GOSUB 910:PRINT
620 FOR X=C+1 TO A:V=M(X):GOSUB 900
625 NEXT: PRINT TAB(30)"["DDDDDDDDD"]"
630 PRINT"   [RV]TOTAL SAVINGS ";
631 PRINT"ON DEPOSIT[RVOFF]... $";
635 V=SD: GOSUB 910
636 FOR X=1 TO 39
637 PRINT"["$",LC,DN,"E",UP]";:NEXT
640 PRINT:PRINT"[DN]^"
641 :
642 REM ============================
643 REM GET USER INPUT & CHECK FOR
644 REM VALID INPUT -
645 REM ============================
646 :
650 PRINT"["-@@"] ACCT#, [RV]P[RVOFF]";
651 INPUT"AY, OR, [RV]D[RVOFF]ONE";A$
660 X=VAL(A$):IF X>0 AND X<=A THEN 800
664 :
665 REM ============================
666 REM ******   CREDIT PAY   ********
667 REM ============================
668 :
670 IFLEFT$(A$,1)<>"P"OR(C1<>0)THEN1000
671 :
672 REM ============================
673 REM GET CHECKING/SAVINGS DEPOSITS
674 REM TO CREDIT STANDARD PAY.
675 REM CAN ONLY USE ONCE PER RUN!
676 REM ============================
677 :
680 INPUT"    CHECKING DEPOSIT";C1
```

```
681 INPUT"    SAVINGS  DEPOSIT";S1
685 C1=INT(100*(C1+.001))
686 S1=INT(100*(S1+.001))
690 FOR X=1 TO A:READ V:V=V*100
695 IF X=C THEN V=C1-C2
700 IF X=A THEN V=S1-S2
710 GOSUB 950:NEXT:GOTO 600
750 :
751 REM ==============================
752 REM *****  CREDIT/DEBIT ACCT *****
753 REM ==============================
754 :
800 PRINT"[DN]AMT TO CREDIT(+)";
805 INPUT" / DEBIT(-)";V
850 V=INT(100*(V+.001)):GOSUB 950
855 GOTO600
890 :
891 REM ==============================
892 REM SUBROUTINE TO GENERATE SINGLE
893 REM LINE OF DISPLAY WITH -
894 REM ACCT #, NAME, AND VALUE IN
895 REM STANDARD FORMAT.
896 REM ==============================
897 :
900 PRINT RIGHT$(STR$(X+100),2);" ";
901 PRINTN$(X);RIGHT$(L$,29-LEN(N$(X)));
905 T$=STR$(INT(ABS(V)/100)*SGN(V))
910 PRINT RIGHT$("    "+T$,4);
915 IFV<0ANDV>-100THENPRINT"[2 LC]-0";
920 PRINT".";RIGHT$(STR$(ABS(V)+100),2)
925 RETURN
940 :
941 REM ==============================
942 REM SUBROUTINE TO CREDIT/DEBIT
943 REM ACCT & UPDATE SAVINGS/CHECKING
944 REM TOTALS.
945 REM ==============================
946 :
950 M(X)=M(X)+V
955 IF X>C THEN SD=SD+V:S2=S2+V:RETURN
960 CB=CB+V:C2=C2+V:RETURN
990 :
991 REM ==============================
992 REM ******   CHECK IF DONE   *****
993 REM ==============================
994 :
1000 IF LEFT$(A$,1) <> "D" THEN 600
1001 :
```

```
1002 REM ============================
1003 REM AT END OF PROGRAM SAVE DATA
1004 REM BACK INTO THE PROGRAM, THEN
1005 REM PRINT REMINDER TO SAVE PGM
1006 REM ============================
1007 :
1010 N=1030:FOR X=1 TO A
1015 L$=RIGHT$("      "+STR$(M(X)),6)
1030 FOR Y=1 TO 6
1035 POKE N,ASC(MID$(L$,Y,1)):N=N+1
1040 NEXT:N=N+1:IF X=C THEN N=N+5
2000 NEXT
2005 PRINT"[CLR]REWIND TAPE AND SAVE "
2006 PRINT"THE PROGRAM"
2010 PRINT"[DN]TO RETAIN THE NEW DATA!"
2011 PRINT"[2 DN]"
2015 END
2020 REM
2030 REM CURSOR POSITIONING AND GRAPHICS
2040 REM CHARACTERS ARE ENCLOSED WITHIN
2050 REM BRACKETS. THE GRAPHIC CHARS.
2060 REM ARE SHOWN AS UNSHIFTED CHARS.
2070 REM BETWEEN QUOTES. THE CURSOR
2080 REM CONTROL CHARACTERS ARE
2090 REM INDICATED AS FOLLOWS:
2100 REM SP=SPACE; LC=LEFT CURSOR
2110 REM UP=UP CURSOR; DN=DOWN CURSOR
2120 REM CLR=CLEAR SCREEN; RV=REVERSE
2130 REM RVOFF=REVERSE OFF
```

Block Access Method Map For A Commodore 2040 Disk Drive

Tom Conrad

A tutorial on some of the complex functions of the Commodore Disk Drives.

Overview

The Block Access Method (BAM) map program will allow you to see where your files are allocated. You can save and delete files and observe the allocation technique.

Description

The purpose of the BAM is to protect allocated files so they are not written over and therefore destroyed. The BAM map resides on the directory track 18. The BAM is in the first half of sector 0.

The layout looks like this:

BAM Dump

TRACK 18 SECTOR 0

	a	b	Track NUMBER
00:	12 01	01 00 15 FF FF 1F	1
08:	15 FF FF 1F	15 FF FF 1F	2 3
10:	15 FF FF 1F	15 FF FF 1F	4 5
18:	15 FF FF 1F	15 FF FF 1F	6 7
20:	15 FF FF 1F	15 FF FF 1F	8 9
28:	15 FF FF 1F	15 FF FF 1F	10 11
30:	15 FF FF 1F	15 FF FF 1F	12 13
38:	15 FF FF 1F	15 FF FF 1F	14 15
40:	15 FF FF 1F	15 FF FF 1F	16 17
48:	12 FC FF 0F	14 FF FF 0F	18 19
50:	14 FF FF 0F	14 FF FF 0F	20 21
58:	14 FF FF 0F	14 FF FF 0F	22 23
60:	14 FF FF 0F		24
		12 FF FF 03	25

```
68: 12 FF FF 03 12 FF FF 03                    26 27
70: 12 FF FF 03 12 FF FF 03                    28 29
78: 12 FF FF 03 ----------------               30
    ----------------
                11 FF FF 01                       31
80: 11 FF FF 01 11 FF FF 01                    32 33
82: 11 FF FF 01 11 FF FF 01                    34 35
```

a - Address of the next sector which is where the directory begins.
b - The start of the BAM map for track 1.

Detailed Explanation

```
          a  b  c  d
00 : 12 01 01 00 15 FF FF 1F
```

a - Total free sectors for track 1. In this case it is hex 15 or decimal 21. Since track 1 has a maximum of 21 sectors, track 1 is totally empty.

b - The bit configurations for sectors 0 through 7. Bit on means empty sector and bit off means allocated sector.

hex FF = bits "1 1 1 1 1 1 1 1"

- 0th sector
- 1th sector
- 2th sector
- 3th sector
- 4th sector
- 5th sector
- 6th sector
- 7th sector

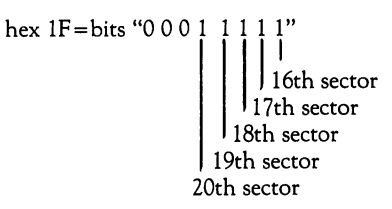
Therefore all sectors are empty.

c - bit configurations for sectors 8 thru 15

hex FF=bits "1 1 1 1 1 1 1 1"

- 8th sector
- 9th sector
- 10th sector
- 11th sector
- 12th sector
- 13th sector
- 14th sector
- 15th sector

d - bit configurations for sectors 16 through 20.



In any empty disk, the "d" byte changes from 1F, 0F, 03, 01 to compensate for varying number of sectors per track.

hex 1F is the pattern where there are 21 sectors as in tracks 1 through 17.

hex 0F is the pattern where there are 20 sectors as in tracks 18 through 24.

hex 1F is the pattern where there are 18 sectors as in tracks 25 through 30.

hex 1F is the pattern where there are 17 sectors in tracks 31 through 35.

Observations Using The BAM Map Program

The BAM turns off the bits when it allocates a sector. The BAM Map Program looks at these bits and if the bit is on (meaning it is free and has not been allocated) it will print either a "▒" or a white square. By looking at the map you can determine how full or empty the disk is.

Varying numbers of sectors.

The reason for the varying number of sectors per track is to pack more data on the disk. The worse case (17 sectors per track) propagated throughout the disk would decrease the number of sectors per track by 95 sectors or 24K.

Sectors not contiguous

The sectors are in 255 byte blocks. A program file, which is stored in 255 bytes, is not written on the disk contiguously, but written approximately one-half track apart. Using the BAM program, you can see when you save a program on an empty disk, that DOS will save the first 255 bytes on track 17 sector 0, the second 255 bytes on sector 10, the third 255 bytes on sector 3, and so on. The purpose of these gaps is to speed up the processing by not waiting for a full rotation of the disk. If the program were written contiguously after each write, DOS would have to wait an entire rotation of the disk to write the next sequential sector. Thus, the BAM Map will show where alternating sectors are allocated.

Allocation of sectors

DOS allocates disk space very efficiently. Sectors are allocated around the directory (track 18). This reduces the read/write head

movement because it reads the directory first, then reads the file. By having the file close to the directory, head movement is reduced.

Where sectors are allocated

When you delete the first program on a full disk, the BAM Map will show free sectors near the directory. When you save a new program, it will start by allocating those free sectors nearest the directory and will start filling in where you deleted the old file. If the new program is larger than the old program, it will try to allocate sectors further and further from the directory. By using this allocation technique, the need for a disk compress is eliminated.

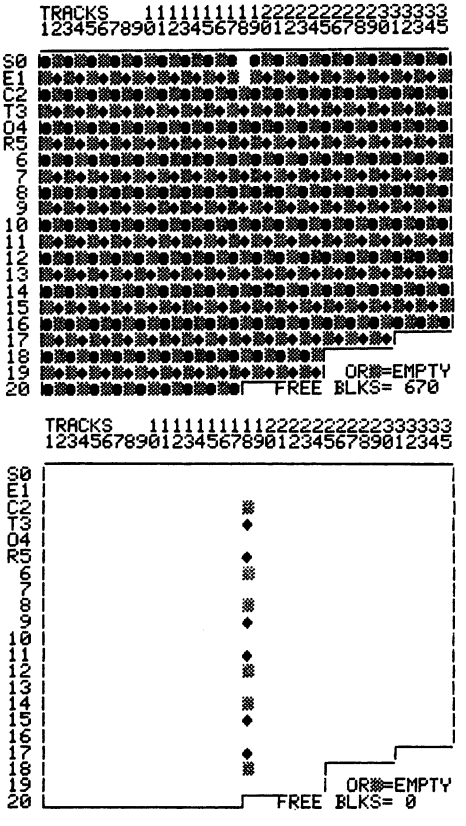
PROGRAM EXPLANATION
100-170 Initialization
180-190 Which drive?
200-430 Prints the BAM Map outline.

```
100 REM* BLOCK ACCESS METHOD DUMP    *
110 REM* WRITTEN BY    TOM CONRAD    *
120 REM*
130 REM*
140 REM*      INITIALIZATION          *
150 DIM A(4)
160 NL$=CHR$(0)
170 T=0: REM TOTAL FREE BLOCKS
175 REM* WHICH DRIVE     *
180 PRINT"ñ↓↓↓↓↓DRIVE?"
190 GET D$: IF D$="" GOTO 190
195 REM* PRINTS THE BAM MAP OUTLINE *
200 PRINT"ñ   rTRACKSî   111111111122222
    ¬22222333333"
210 PRINT"   12345678901234567890123456
    ¬89012345"
220 PRINT"$  $$$$$$$$$$$$$$$$$$$$$$$$$$
    ¬$$$$$$$$"
230 PRINT"rSî0'                          ¬
    ¬          $"
240 PRINT"rEî1'                          ¬
    ¬          $"
250 PRINT"rCî2'                          ¬
    ¬          $"
260 PRINT"rTî3'                          ¬
    ¬          $"
270 PRINT"rOî4'                          ¬
    ¬          $"
```

```
280 PRINT"rRî5'                                    ¬
    ¬              &"
290 PRINT" 6'                                      ¬
    ¬           &"
300 PRINT" 7'                                      ¬
    ¬           &"
310 PRINT" 8'                                       ¬
    ¬           &"
320 PRINT" 9'                                       ¬
    ¬           &"
330 PRINT"10'                                       ¬
    ¬           &"
340 PRINT"11'                                       ¬
    ¬           &"
350 PRINT"12'                                       ¬
    ¬           &"
360 PRINT"13'                                       ¬
    ¬           &"
370 PRINT"14'                                       ¬
    ¬           &"
380 PRINT"15'                                       ¬
    ¬           &"
390 PRINT"16'                                       ¬
    ¬           &"
400 PRINT"17'                                        ¬
    ¬   O####"
410 PRINT"18'                                  ¬
    ¬O#####"
420 PRINT"19'                                 &r ¬
    ¬îOR&=EMPTY"        "
430 PRINT"20'$$$$$$$$$$$$$$$$$O######"
440 S$="": T$=""
450 W$="ϕ": FOR I=1 TO 25: S$=S$+W$:NEXT
460 V$="→": FOR I=1 TO 40: T$=T$+V$:NEXT
465 REM* INIT DRIVE AND CK FOR ERROR *
470 OPEN 15,8,15,"I"+D$:GOSUB 760
475 REM* ALLOC BUFFER Ø TO CHANNEL 2 *
480 OPEN 2,8,2,"#"+"Ø": GOSUB 760
485 REM* BLOCK-READ INTO BUFFER      *
490 PRINT#15,"U1:2,"D$,18,Ø: GOSUB 760
495 REM* SET BUFFER POINTER          *
500 PRINT#15,"B-P:2,4"
505 REM* MEMORY READ                 *
510 PRINT#15,"M-R"CHR$(Ø)CHR$(17)
520 REM* SEARCH FOR EMPTY SECTORS *
530 FOR I=1 TO 35
```

```
540 :FOR L=1 TO 4
545 ::REM*  GETS A BYTE FROM BUFFER    *
550 ::GET#2,A$
560 ::IF A$="" THEN A$=NL$
565 ::REM*CONVERSION FROM CHAR TO ASCI
570 ::A(L)= ASC(A$)
580 ::IF L=1 AND I<>18 THEN T=T+A(1)
590 :NEXT L
600 :FOR J=2 TO 4
605 ::REM* PRINTS ALTERNATING SQUARES  *
610 ::PRINT "h&":IF INT(J/2)=J/2 THEN ¬
    ¬PRINT "hr "
620 ::IF A(J)=ASC(CHR$(0)) THEN GOTO  ¬
    ¬680:REM* SECT FULL *
630 ::FOR K=7 TO 0 STEP -1
635 :::REM* PRINTS ALTERNATING SQUARES  ¬
    ¬*
640 :::PRINT "h&":IF INT(K/2)=K/2 THEN ¬
    ¬PRINT "hr "
645 :::REM* DECODES DECIMAL TO BIT   *
650 :::IF (A(J)-2^K)<0 GOTO 670
660 :::A(J)=A(J)-2^K:GOSUB 790
670 ::NEXT K
680 :NEXT J
690 NEXT I
700 PRINT"h ":REM* CLEAR SQUARE *
705 REM* PRINTS TOTAL FREE BLOCKS     *
710 PRINT LEFT$(S$,22)LEFT$(T$,23)"FREE ¬
    ¬BLKS="T"↑↑↑↑"
715 REM* MAP ON SCREEN UNTIL KEY IS HIT*
720 GET Z$:IF Z$="" GOTO 720
730 CLOSE 2:CLOSE 15
735 REM* START PROGRAM AGAIN  *
740 GOTO 170
750 REM* CHECK FOR DISK ERROR *
760 INPUT#15,EN$,EM$,ET,ES: IF EN$="00" ¬
    ¬THEN RETURN
770 PRINT "rDISK ERROR:î " EM$ " " EN$,
    ¬ET "," ES
780 END
790 REM* PRINT ALTERNATING PATTERN *
800 IF INT(I/2)= I/2 AND INT(K/2)= K/2 ¬
    ¬THEN C$="&"
810 IF INT(I/2)<>I/2 AND INT(K/2)= K/2 ¬
    ¬THEN C$="r "
820 IF INT(I/2)= I/2 AND INT(K/2)<>K/2 ¬
    ¬THEN C$="r "
```

```
830 IF INT(I/2)<>I/2 AND INT(K/2)<>K/2 ¬
      ¬THEN C$="r&"
840 PRINT "h" LEFT$(S$,3+((J-2)*8)+K) ¬
      ¬LEFT$(T$,2+I) C$
850 RETURN
```



References:

Parsons, James C., "Display Track and Sector", Commodore Newsletter Vol. 1 Number 8, January 1980.

Commodore Business Machines, Commodore CBM Dual Floppy Disk Model 2040 User Manual, July 1979.

440-460 Sets up S$ with 25 cursor downs. Sets up T$ with 40 cursor rights.
470 Initializes the requested drive & checks for a disk error.
480 Allocates buffer 0 to channel 2 for block commands that follow.

490 User command that does a block-read. It reads from the requested disk, track 18, sector 0 into the disk buffer and checks for a disk error.
500 Set the channel 2 pointer to the 5th byte in the buffer where the BAM Map starts.
510 Memory-Read Command sets up the byte pointed to by the address 1700.
530-690 Read the BAM and look for empty sectors.
550 The GET# will receive one byte from the buffer via channel 15
560 The byte is in character form and, if it is null, it needs to be changed to CHR$(0) otherwise statement 570 will end the conversion.
570 Conversion to numeric.
580 This calculates the total free sectors available. A(1) is the total free sectors for that particular track. The total is calculated by summing all the A(1)'s except Track 18 which is reserved only for the directory and cannot be allocated for any files.
610 Prints in the upper left corner a alternating▒and white squares to show when the program is running.
620 If sector is full (all bits are off) go to next byte.
630-670 Decodes the decimal number into bit pattern and check if bit is on.
640 Same as 610
650 If the number minus the powers of 2 is greater than or equal to zero then the bit is on and go to 790 to print▒on the screen.
700 Program is finished running now clear the square in the upper left corner.
710 Print the total free blocks.
720 Keep map on screen until any key is typed.
730 Close the files.
740 Start program again.
750-780 Subroutine to check for disk error.
790-850 Prints▒or white square for the empty sector.
840 The first LEFT$ is calculating how many cursor downs (sector no.) are needed and the second one is calculating how many cursor LEFT$ (track no.) are needed.

Disk Lister

A Disk Cataloging Program For The Commodore PET And 2040 Disk

Baker Enterprises

Having finally copied all my programs from cassette onto floppy disks, I suddenly found it somewhat difficult to find out where anything was. With well over 300 programs scattered on to 20 or 30 disks, it just wasn't easy to quickly locate a particular program. In addition, I was starting to use Word Pro 3 quite heavily to write articles and various documents, saving them all on disk as well. Because of this, I decided to write a program to catalog all the disks and condense the information onto a single diskette.

The program shown here is the first step toward my final goal. It can catalog well over 100 diskettes with the current Commodore 2040 disk drive. It only has a few functions implemented, but it has proven to be very handy. I have a "wish" list of other features I intend to add in the near future. All I need now is the time to do it!

The major flow of the program should be straightforward. I've sprinkled the program with REMarks to help document several operations and a few of the variables used. If you should copy the program, I would strongly recommend leaving out all REMarks and unnecessary spaces to help speed up program execution.

In its present form, the program reads the directory of any disk placed in drive #1. It then writes a condensed directory as a data file on the master directory disk drive #0. All of this is done automatically without any user input other than selecting the program function and verifying that the correct disk was inserted. Once the data files are created, you can then display or print the directory of any disk that has been cataloged in the master directory. The directory will show the disk name, ID, and format. It will also show an alphabetized list of the files on the disk along with the file type and length (in blocks) of each file. While a directory is being listed, hitting "S" will stop the listing until another key is hit. Hitting "Q" at any time during the listing will terminate the list function. A sample directory printout is shown in Figure 1 to give you an idea of what is displayed.

The file names of the sequential data files created for the master directory consist of the two character disk ID followed by a period and the letters DIR. In its compacted form, the major disk information takes 25 bytes and each entry in the directory takes 20 bytes. Since the disk ID is used to create the data file name, be careful not to duplicate disk IDs. This precaution is also recommended when upgrading to DOS 2.0 since DOS uses the ID to recognize that a disk has been changed in the drive. Another hint on using this program — reserve one disk as the master directory disk with nothing else stored on that disk except the directory data files. This will allow cataloging the maximum number of disks into your master directory.

If a catalogued disk is later updated or modified, simply re-catalog the disk to update the master directory. The old data file will be deleted and a new one created, all automatically. The program also provides a delete function, so you can delete a cataloged disk that no longer exists. This function simply deletes the appropriate data file for the specified disk ID. You could actually accomplish the same function by manually scratching the correct data file from the master directory disk.

Currently, when listing or deleting directories, you must enter the two character disk ID. This can be inconvenient at times, but it does make things easier. I intend to allow entering the ID *or* the disk name in the next version I'm working on. However, this will require maintaining some kind of cross-reference to correlate the disk IDs and disk names. When this feature is added, the delete function will always have to be used to remove a disk from the master directory. The added cross-reference will also be the basis for several other features I intend to add:

List all disk IDs currently used in alphabetical order; optionally display each disk's corresponding 16 character name. This will help avoid using duplicate disk IDs when creating new disks.

List all disk names in alphabetical order and show each disk's corresponding 2-character disk ID.

Ability to list all disks on which a particular file can be found. This function should use character matching in case you can't remember the exact file name or want all files starting with a particular word, etc.

One other thing I would like to add is computation of the number of free blocks from the BAM. If this information were included in

the data files for each disk, you could then list all disks with the number of free blocks displayed. This would allow quickly finding space on a disk to save a new program of known length.

Right now I'm not sure when I'll be able to get around to finishing this project. At least I've got something useful for now and it does help tremendously. If you have any ideas or suggestions as to other features you think might be useful, or if you're interested in how the final version turns out, let me know.

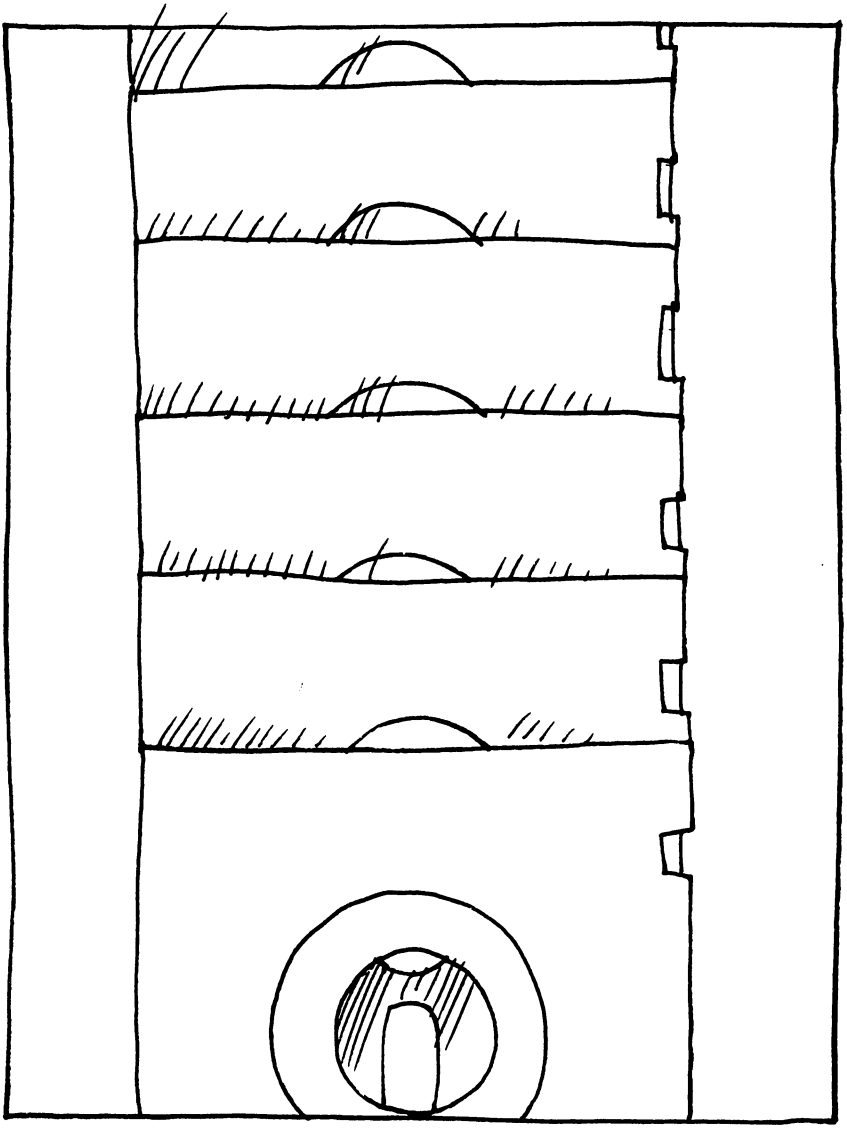
```
10 REM ******* DISK LISTER ************
20 REM
30 REM      BY: ROBERT W. BAKER
40 REM
50 REM 15 WINDSOR DRIVE, ATCO, NJ 08004
60 REM
70 REM *********************************
80 :
90 CLR:DIM D$(150),D(150):Q$=CHR$(34):
     ¬CR$=CHR$(13)
100 REM DISPLAY MENU & SELECT FUNCTION
110 PRINT"ñ";SPC(9);"rD I S K   L I S T ¬
     ¬E R":GOSUB 1340
120 PRINT SPC(5);"0 - DONE↓
130 PRINT SPC(5);"1 - UPDATE MASTER ¬
     ¬DIRECTORY↓
140 PRINT SPC(5);"2 - DISPLAY SELECTED ¬
     ¬DIRECTORY↓
150 PRINT SPC(5);"3 - DELETE DISK ENTRY ¬
     ¬FROM MASTER
160 GOSUB 1340
170 PRINT"↓ENTER DESIRED FUNCTION:  ";
180 GOSUB 1320
190 IF C$="0" THEN PRINT"ñ": END
200 C=VAL(C$):IF C<1 OR C>3 THEN 180
210 ON C GOTO 250,750,1050
220 REM ************************
230 REM UPDATE MASTER DIRECTORY
240 REM ************************
250 PRINT"ñINSERT UPDATE DISK IN DRIVE ¬
     ¬#1
260 GOSUB 1310:GOSUB 1340:PRINT"OK
270 OPEN 15,8,15
280 PRINT#15,"I1"
290 OPEN 5,8,5,"$1,S,R":GOSUB 1260
300 Y=142:GOSUB 1200:REM *** SKIP BAM
310 Y=16:GOSUB 1180:DN$=S$:REM *** DISK ¬
     ¬NAME
```

```
320 Y=2:GOSUB 1200:REM *** SKIP SPACES
330 Y=2:GOSUB 1180:DI$=S$:REM *** DISK ¬
    ¬ID
340 PRINT"ĥrDISK NAME:f̂  ";DN$ :
    ¬ PRINT"ŧrDISK  ID:f̂   "DI$:
    ¬GOSUB 1340
350 PRINT"CORRECT DISK INSERTED"; :
    ¬ GOSUB 1350:IF C$="N" THEN 710
360 GOSUB 1340:PRINT"READING DIRECTORY ¬
    ¬ENTRIES...
370 GOSUB 1250
380 Y=2:GOSUB 1180:DF$=S$:REM *** DISK ¬
    ¬FORMAT
390 Y=89:GOSUB 1200:NF=0:Z=0:REM *** ¬
    ¬SKIP TO FIRST DIRECTORY ENTRY
400 GOSUB 1220:FT=V:F$=C$:REM *** FILE ¬
    ¬TYPE (0=DELETED)
410 Y=2:GOSUB 1200:REM *** SKIP ¬
    ¬STARTING TRACK & SECTOR
420 Y=16:GOSUB 1180:REM *** FILE NAME
430 Y=9:GOSUB 1200:REM *** SKIP UNUSED ¬
    ¬INFO
440 GOSUB 1220:X=V:GOSUB 1220:X=X+(V*256
    ¬):REM *** #BLOCKS IN FILE
450 IF FT>0 THEN NF=NF+1:D$(NF)=F$+S$:
    ¬D(NF)=X:REM *** ADD FILE IF NOT ¬
    ¬DELETED
460 Z=Z+1:Z=Z-(INT(Z/8)*8):REM *** ¬
    ¬Z=ENTRY WITHIN THIS DISK BLOCK
470 IF Z>0 THEN Y=2:GOSUB 1200:REM ** ¬
    ¬SKIP 2 BYTES IF NOT LAST ENTRY IN ¬
    ¬BLOCK
480 IF SS=0 THEN 400:REM *** CONTINUE ¬
    ¬TILL END OF DIRECTORY
490 CLOSE 5:IF NF<2 THEN 600
500 GOSUB 1340
510 PRINT"SORTING DIRECTORY ENTRIES...
520 REM SORT DIRECTORY INTO
530 REM ALPHABETICAL ORDER
540 FOR X=1 TO NF:FOR Y=1 TO NF-1
550 IF D$(Y)<=D$(Y+1) THEN 570
560 C$=D$(Y):C=D(Y):D$(Y)=D$(Y+1):
    ¬D(Y)=D(Y+1):D$(Y+1)=C$:D(Y+1)=C
570 NEXT Y,X
580 REM DELETE OLD DIRECTORY
590 REM DATA FILE & SAVE NEW COPY
600 GOSUB 1340:PRINT"UPDATING MASTER ¬
    ¬DIRECTORY...
```

```
610 S$="0:"+DI$+".DIR"
620 PRINT#15,"S"+S$
630 OPEN 5,8,5,S$+",S,W":GOSUB 1260
640 PRINT#5,Q$;DN$;Q$;CR$;:GOSUB 1260
650 PRINT#5,DI$;CR$;:GOSUB 1260
660 PRINT#5,DF$;CR$;:GOSUB 1260
670 IF NF=0 THEN 710
680 FOR X=1 TO NF:FOR Y=1 TO 17:
      ¬PRINT#5,MID$(D$(X),Y,1);:GOSUB ¬
      ¬1260:NEXT Y
690 H=INT(D(X)/256):L=D(X)-(256*H)
700 PRINT#5,CHR$(L);CHR$(H);CR$;:
      ¬GOSUB 1260:NEXT X
710 CLOSE 5:CLOSE 15:GOTO 110
720 REM ********************************
730 REM DISPLAY SELECTED DISK DIRECTORY
740 REM ********************************
750 PRINT"ñTO DISPLAY DISK DIRECTORY":
      ¬GOSUB 1140:OPEN 15,8,15
760 OPEN 5,8,5,S$+",S,R":GOSUB 1260
770 GOSUB 1340:PRINT"WANT PRINTED ¬
      ¬COPY";:GOSUB 1350:GOSUB 1340
780 PD=3:IF C$="Y" THEN PD=4
790 OPEN 4,PD:REM *** PD = PRINT DEVICE ¬
      ¬SELECTOR (3=DISPLAY, 4=PRINTER)
800 INPUT#5,DN$:GOSUB 1260
810 INPUT#5,DI$:GOSUB 1260
820 INPUT#5,DF$:GOSUB 1260
830 IF PD=3 THEN PRINT"ñ";
840 PRINT#4,"rDISK NAME:î  ";DN$
850 PRINT#4
860 PRINT#4,"rDISK ID:î  ";DI$;SPC(10);"
      ¬rDISK FORMAT:î  ";DF$
870 PRINT#4 : REM *** DISK FORMAT WILL ¬
      ¬BE BLANK FOR DOS 1.0
880 PRINT#4,"CCCCCCCCCCCCCCCCCCCCCCCCCCC
      ¬CCCCCCCCCCCC":PRINT#4
890 Y=17:GOSUB 1180:REM *** GET FILE ¬
      ¬NAME & TYPE
900 GOSUB 1220:Z=V:GOSUB 1220:Z=Z+(256*V
      ¬):REM *** GET #BLOCKS
910 GOSUB 1250:REM *** SKIP LAST CR
920 PRINT#4,RIGHT$("    "+STR$(Z),
      ¬4);"  ";
930 PRINT#4,MID$(S$,2,16);SPC(3);
940 V=ASC(LEFT$(S$,1)):REM *** DECODE ¬
      ¬FILE TYPE
950 IF V=129 THEN PRINT#4,"SEQ";
```

```
960 IF V=130 THEN PRINT#4,"PGM";
970 IF V=131 THEN PRINT#4,"USR";
980 PRINT#4:GET C$:IF C$="S" THEN GOSUB ¬
    ¬1320:REM *** ALLOW START/STOP OF ¬
    ¬LIST
990 IF C$<>"Q" AND SS=0 THEN 890
1000 CLOSE 4:CLOSE 5:CLOSE 15:IF PD=3 ¬
    ¬THEN GOTO 1300
1010 GOTO 110
1020 REM ******************************
1030 REM DELETE DISK DIRECTORY DATA FILE
1040 REM ******************************
1050 PRINT"ñTO DELETE DISK FROM MASTER ¬
    ¬DIRECTORY":GOSUB 1140:OPEN 15,8,15
1060 PRINT#15,"S"+S$:CLOSE 15:GOTO 110
1070 :
1080 REM ***********************
1090 REM ***** SUBROUTINES *****
1100 REM ***********************
1110 :
1120 REM *** GET DISK ID
1130 REM *** & MAKE DATA FILE NAME
1140 INPUT"ÿENTER DISK ID   "<<<";DI$
1150 IF DI$="" THEN 110
1160 S$="0:"+LEFT$(DI$,2)+".DIR":RETURN
1170 REM *** READ STRING FROM DISK,
    ¬ Y-BYTES LONG
1180 S$="":FOR X=1 TO Y:GOSUB 1250:
    ¬S$=S$+C$:NEXT X:RETURN
1190 REM *** SKIP Y-BYTES OF DISK FILE
1200 FOR X=1 TO Y:GOSUB 1250:NEXT X:
    ¬RETURN
1210 REM *** READ BYTE & RETURN ASC ¬
    ¬VALUE
1220 V=0:GOSUB 1250:IF C$<>"" THEN ¬
    ¬V=ASC(C$)
1230 RETURN
1240 REM *** GET BYTE & CHK FOR DISK ¬
    ¬ERROR
1250 GET#5,C$:SS=ST
1260 INPUT#15,EN,EM$,ET,ES:IF EN=0 THEN ¬
    ¬RETURN
1270 PRINT"ñrDISK ERROR!ÿ
1280 PRINT EN;EM$;ET;ES:CLOSE 4:CLOSE 5:
    ¬CLOSE 15
1290 REM *** MISC ROUTINES ***
1300 GOSUB 1340:GOTO 110
1310 PRINT"ÿDEPRESS ANY KEY TO CONTINUE
```

```
1320 GET C$:IF C$="" THEN 1320
1330 RETURN
1340 PRINT"↓@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
     ¬@@@@@@@@@@@@":RETURN
1350 PRINT" (Y/N) ?  ";
1360 GOSUB 1320:IF C$<>"Y" AND C$<>"N" ¬
     ¬THEN 1360
1370 PRINT C$ : RETURN
```



Compactor

Robert W. Baker

This program is invaluable to programmers who cringe when they "run out of memory."

This program is the result of several days of experimenting with BASIC program structures and the 2040 disk. In short, the program will read a BASIC program that was saved on disk and create a new, compacted copy. The program will delete all REMarks, unnecessary spaces, and leading colons. Much of this is similar to other utility programs currently available. However, this program goes one step further. It combines program lines whenever possible to eliminate the link, line number, and line-end-flag overheads normally associated with each line. It will make a program as small as possible, and most likely faster running.

While creating this program, I came across a few undocumented "quirks" of Commodore BASIC. Since many people are currently experimenting with the capabilities of having programs "write" programs on disk, this information may be of interest:

Zero Length Lines:

Normally, it is impossible to create a zero length line using the screen editor on the PET. By zero length line, I mean a line with a link, line number, and end-of-line flag, but no BASIC commands or text. If you were to type just a line number using the screen editor, you would actually delete a line instead of entering a zero length line. However, when writing a BASIC program on disk as a data file, there is nothing stopping you from entering a zero length line. But if you want the program to run, you cannot have any zero length lines in the program. BASIC cannot link the program lines correctly whenever there is a zero length in the program.

Long Lines:

At the other extreme, you cannot create a BASIC line that is longer than 255 bytes. Again, using the PET screen editor you could not create such a line. You are normally limited to a maximum of 78 bytes due to the line wrapping characteristics and at least a one digit line number. When writing a BASIC program on disk as a data file, be careful not to create a line greater than 255 bytes. Otherwise the program will usually not load from the disk. If it does load, the program will be totally destroyed and unusable.

Printing Long Lines:

Here's a quick comment on the Commodore printers. If you list a program that contains lines longer than 80 characters, the printed listing may be incorrect. It appears that the printer occasionally switches out of listing mode and into print mode when a line exceeds 80 characters. At the start of the next line everything is ok again.

Program Description

When running the COMPACTOR program, the BASIC program to be compacted must be on the diskette inserted in drive #0. The new compacted version will be written on the diskette in drive #1 with the same filename, but with a "/C" suffix. The program will read the program to be compacted as a sequential disk data file, and the file will be read twice.

The first pass checks for line numbers within the subject program that are targets of: GOTO, GOSUB, or IF . . . THEN (line #) statements. When a target line is found, it is saved in matrix TL if not already saved. A check is also made for multiple target lines in ON . . GOTO and ON . . GOSUB statements. Each target line will be displayed on the PET screen in the order found. This helps to give some indication of the scanning progress, since it can be rather slow.

During the second pass, each line is copied, deleted, or compacted as appropriate. The line number will be displayed as each line is processed to let you know how the program is progressing. The rules followed by the COMPACTOR are as follows:

Any leading colons and/or spaces on a line are deleted.

A line that has only REMarks is deleted if it is not a target line. The remark will be replaced with a single colon if the line is a target line and must be retained. This may produce a leading colon if the next line is not a target line and is combined with this line. The line cannot be reduced to a zero length line since BASIC cannot link a program correctly with a zero length line, as mentioned earlier.

If any line contains an IF . . . THEN or GOTO statement, another line cannot be combined with this line. Any line combined after these BASIC commands would never be executed, thus the program would not function properly.

Any spaces within a line, not enclosed in quotes, are deleted.

Any REMarks at the end of a BASIC line are deleted to the end of the line.

Anything within quotes is copied, untouched. If an ending quote is missing from the line, one is added if another line could be combined with this line. Therefore, if a line does not contain an IF . . . THEN or GOTO statement, an ending quote is added.
When a colon is found within a BASIC line and not within quotes, the next non-space character is checked before copying the colon. If a REMark follows the colon, the colon and the rest of the line is deleted. Otherwise the colon is copied, and processing continues as normal.
At the end of each BASIC line, a check is made to see if the next line can be combined with this line. If there were no IF . . . THEN or GOTO commands, and the next line is not a target line, the lines are combined. When combining lines, the line and line number are discarded, a colon is written, and the next line is processed as part of the previous line.
If the next line cannot be combined with the current line, the end of line flag is copied along with a dummy link and the next line number. A dummy link is used to avoid excesive processing and working buffers necessary with calculating program links. Also, the links are automatically corrected by PET BASIC with the RUN or CLR commands. As a standard operating procedure, the newly created program output by COMPACTOR should be loaded and re-linked, then re-saved onto disk. The program can be re-linked by issuing a CLR command after being loaded.
As mentioned previously, a BASIC program line cannot exceed 255 bytes in length. If it does, the program may not load from disk or it may be totally unusable. To protect against this, the COMPACTOR program stops combining lines if more than 170 bytes have been written in a single basic line. Since any normal line cannot exceed 78 bytes in length, this should insure that no program generated lines are longer than the maximum length.

As an example of what this program will do, I included a listing of a compacted version of the COMPACTOR program itself. Since this program has many REMarks, compacting saves over 3000 bytes for about a 50% saving in memory space. For most programs, the savings will be much smaller, depending on programming style. A side benefit, however, is the increase in the operating speed of compacted programs. I should warn, though, that the compacting process can be rather slow. Compacting of the COMPACTOR program (a 6K program with all the REMarks) takes about 16 minutes. But all you have to do is start if off and then go get a cup of coffee while the PET does the work! And you only have to run it

once for any given program!

```
10 REM ****************************
20 REM *    C O M P A C T O R     *
30 REM *  --------------------    *
40 REM *    BY:  ROBERT BAKER     *
50 REM *                          *
60 REM *    BAKER ENTERPRISES     *
70 REM *     15 WINDSOR DR.       *
80 REM *    ATCO, N.J.  08004     *
90 REM ****************************
100 :
110 CLR : DIM TL(1000)
120 :
130 REM ****************
140 REM READY DISK FILES
150 REM ****************
160 :
170 PRINT"ñ"SPC(15)"rCOMPACTOR↓↓
180 PRINT" rINPUTî FILE IN rDRIVE #0↓
190 PRINT"rOUTPUTî FILE IN rDRIVE #1↓↓
200 INPUT"rINPUT FILE NAMEî";FL$
210 PRINT"ñSCANNING FILE
220 PRINT"   FOR TARGET LINES.....↓↓
230 OPEN 15,8,15 : GOSUB 2370
240 OPEN 5,8,5,"0:"+FL$+",P,R"
250 :
260 REM ****************************
270 REM READ LOAD ADR, LINK & LINE#
280 REM ****************************
290 :
300 GOSUB 2370 : GOSUB 2310
310 GOSUB 2310 : IF V+V1=0 THEN 790
320 GOSUB 2310 : LN=V1+(256*V)
330 :
340 REM *****************************
350 REM         SCAN BASIC LINES
360 REM FOR GOTO, GOSUB & THEN TOKENS
370 REM *****************************
380 :
390 GOSUB 2330
400 IF V=0 THEN 310
410 IF V=137 OR V=141 THEN 480
420 IF V<>167 THEN 390
430 :
440 REM ****************
450 REM GET TARGET LINE#
460 REM ****************
470 :
480 LT=0
490 GOSUB 2330 : IF V=32 THEN 490
500 IF V<48 OR V>57 THEN 580
```

```
510 LT=(10*LT)+VAL(C$)
520 GOSUB 2330 : GOTO 500
530 :
540 REM ************************
550 REM CHECK IF ALL READY FOUND
560 REM ************************
570 :
580 FOR X=0 TO N
590 IF TL(X)=LT THEN 710
600 NEXT X
610 TL(N)=LT : N=N+1
620 PRINT LT,
630 IF N<1000 THEN 710
640 PRINT"↓↓TOO MANY TARGET LINES!
650 GOTO 2430
660 :
670 REM ****************************
680 REM CHECK FOR 'ON...GOTO/GOSUB'
690 REM ****************************
700 :
710 IF V=44 THEN 480
720 IF V<>32 THEN 400
730 GOSUB 2330 : GOTO 710
740 :
750 REM *****************
760 REM SORT TARGET LINES
770 REM *****************
780 :
790 IF N<2 THEN 900
800 FOR X=0 TO N-1
810 FOR Y=0 TO N-2
820 IF TL(Y) < TL(X) THEN 840
830 V=TL(Y) : TL(Y)=TL(X) : TL(X)=V
840 NEXT Y,X
850 :
860 REM *********************
870 REM GET READY FOR COMPACT
880 REM *********************
890 :
900 PRINT "ñCOMPACTING LINES....↓↓
910 CLOSE 5
920 OPEN 5,8,5,"0:"+FL$+",P,R"
930 GOSUB 2370
940 FO$=LEFT$(FL$,14)+"/C"
950 PRINT#15,"S1:"+FO$
960 OPEN 6,8,6,"1:"+FO$+",P,W"
970 GOSUB 2370
980 :
990 REM  **************
1000 REM COPY LOAD ADR
1010 REM **************
1020 :
```

```
1030 GOSUB 2310
1040 PRINT#6,CHR$(V1);
1050 PRINT#6,CHR$(V); : R=0
1060 :
1070 REM ************************
1080 REM COPY LINK & LINE NUMBER
1090 REM ************************
1100 :
1110 GOSUB 2310 : K1=V1 : K2=V
1120 F=0 : IF V+V1=0 THEN 2230
1130 GOSUB 2310 : L1=V1 : L2=V
1140 LN=L1+(256*L2) : PRINT LN,
1150 GOSUB 2330
1160 IF V=32 OR V=58 THEN 1150
1170 IF V=0 THEN 1200
1180 IF V<> 143 THEN 1240
1190 GOSUB 2330 : IF V>0 THEN 1190
1200 F=1 : FOR X=0 TO N
1210 IF TL(X)<LN THEN NEXT X
1220 IF TL(X)=LN THEN 1240
1230 GOTO 1110
1240 PRINT#6,CHR$(K1);CHR$(K2);
1250 PRINT#6,CHR$(L1);CHR$(L2); : R=4
1260 IF F THEN PRINT#6,":"; : R=5
1270 F=0 : GOTO 1360
1280 :
1290 REM ***************************
1300 REM ****  SCAN BASIC LINE  ***
1310 REM **** & COMPACT PROGRAM ***
1320 REM ***************************
1330 :
1340 PRINT#6,C$; : R=R+1
1350 GOSUB 2330
1360 IF V=137 THEN F=1
1370 IF V=139 OR V=167 THEN F=1
1380 IF V=0 THEN 1820
1390 IF V=32 THEN 1350
1400 :
1410 REM ********************
1420 REM 'REM' TOKEN -
1430 REM DISCARD REST OF LINE
1440 REM ********************
1450 :
1460 IF V<>143 THEN 1550
1470 GOSUB 2330 : IF V>0 THEN 1470
1480 GOTO 1820
1490 :
1500 REM ************************
1510 REM QUOTE -
1520 REM COPY TILL NEXT OR LINE END
1530 REM ************************
1540 :
```

```
1550 IF V<>34 THEN 1690
1560 PRINT#6,C$; : R=R+1
1570 GOSUB 2330
1580 IF V=34 THEN 1340
1590 IF V>0 THEN 1560
1600 IF F THEN V=0 : GOTO 1050
1610 PRINT#6,CHR$(34); : R=R+1
1620 GOTO 1820
1630 :
1640 REM ************************
1650 REM IF COLON - CHK NEXT CHAR
1660 REM       ELSE COPY CHAR
1670 REM ************************
1680 :
1690 IF V<>58 THEN 1340
1700 GOSUB 2330
1710 IF V=32 OR V=58 THEN 1700
1720 IF V=143 THEN 1470
1730 IF V=0 THEN 1820
1740 PRINT#6,":"; : R=R+1
1750 GOTO 1360
1760 :
1770 REM ****************************
1780 REM END OF LINE -
1790 REM CAN WE COMPACT THESE LINES ?
1800 REM ****************************
1810 :
1820 IF F OR (R>170) THEN V=0:GOTO 1050
1830 GOSUB 2310
1840 IF V+V1=0 THEN 2230
1850 GOSUB 2310 : LN=V1+(256*V)
1860 L1=V1 : L2=V : PRINT LN,
1870 :
1880 REM ************************
1890 REM CHK IF LINE# IS A TARGET
1900 REM ************************
1910 :
1920 FOR X=0 TO N
1930 IF TL(X)<LN THEN NEXT X
1940 IF TL(X)=LN THEN 2110
1950 :
1960 REM ********************
1970 REM NOT USED -
1980 REM DISCARD LINK & LINE#
1990 REM ********************
2000 :
2010 GOSUB 2330 : IF V=143 THEN 1470
2020 IF V=32 OR V=58 THEN 2010
2030 IF V=0 THEN 1830
2040 PRINT#6,":"; : R=R+1 : GOTO 1360
2050 :
2060 REM ****************************
```

```
2070 REM LINE# NEEDED -
2080 REM WRITE LINE END, LINK & LINE#
2090 REM ****************************
2100 :
2110 PRINT#6,CHR$(0);CHR$(1);CHR$(1);
2120 PRINT#6,CHR$(L1);CHR$(L2); : R=4
2130 GOSUB 2330
2140 IF V=32 OR V=58 THEN 2130
2150 IF V=0 OR V=143 THEN PRINT#6,":";
2160 F=0 : GOTO 1360
2170 :
2180 REM ********************
2190 REM END OF COMPACT -
2200 REM WRITE END OF PROGRAM
2210 REM ********************
2220 :
2230 PRINT#6,CHR$(0);CHR$(0);CHR$(0);
2240 PRINT"ñrDONE↓↓"
2250 GOTO 2430
2260 :
2270 REM *************************
2280 REM *****  SUBROUTINES  *****
2290 REM *************************
2300 :
2310 GOSUB 2330 : V1=V
2320 :
2330 GET#5,C$ : GOSUB 2370
2340 IF C$="" THEN V=0 : RETURN
2350 V=ASC(C$) : RETURN
2360 :
2370 INPUT#15,EN,EM$,ET,ES
2380 IF EN=0 THEN RETURN
2390 :
2400 PRINT : PRINT"↓↓↓rDISK ERROR↓
2410 PRINT EN;EM$;ET;ES
2420 :
2430 CLOSE 5 : CLOSE 6 : CLOSE 15
READY.
```

```
110 CLR:DIMTL(1000):PRINT"ñ"SPC(15)"rCOMPA
      ¬CTOR↓↓":PRINT" rINPUTî FILE IN ¬
      ¬rDRIVE #0↓":PRINT"rOUTPUTî FILE IN ¬
      ¬rDRIVE #1↓↓":INPUT"
rINPUT FILE NAMEî";FL$:PRINT"ñSCANNING ¬
      ¬FILE":PRINT"   FOR TARGET LINES.....
      ¬↓↓"
230 OPEN15,8,15:GOSUB2370:OPEN5,8,5,"0:
      ¬"+FL$+",P,R":GOSUB2370:GOSUB2310
310 GOSUB2310:IFV+V1=0THEN790
320 GOSUB2310:LN=V1+(256*V)
390 GOSUB2330
```

```
400 IFV=0THEN310
410 IFV=137ORV=141THEN480
420 IFV<>167THEN390
480 LT=0
490 GOSUB2330:IFV=32THEN490
500 IFV<48ORV>57THEN580
510 LT=(10*LT)+VAL(C$):GOSUB2330:GOTO500
580 FORX=0TON:IFTL(X)=LTTHEN710
600 NEXTX:TL(N)=LT:N=N+1:PRINTLT,:
      ¬IFN<1000THEN710
640 PRINT"↓↓TOO MANY TARGET LINES!":
      ¬GOTO2430
710 IFV=44THEN480
720 IFV<>32THEN400
730 GOSUB2330:GOTO710
790 IFN<2THEN900
800 FORX=0TON-1:FORY=0TON-2:IFTL(Y)<TL(X)T
      ¬HEN840
830 V=TL(Y):TL(Y)=TL(X):TL(X)=V
840 NEXTY,X
900 PRINT"ñCOMPACTING LINES....↓↓":CLOSE5:
      ¬OPEN5,8,5,"0:"+FL$+",P,R":GOSUB2370:
      ¬FO$=LEFT$(FL$,14)+"/C":PRINT#15,"S1:
      ¬"+FO$:OPEN6,8
,6,"1:"+FO$+",P,W":GOSUB2370:GOSUB2310:
      ¬PRINT#6,CHR$(V1);
1050 PRINT#6,CHR$(V);:R=0
1110 GOSUB2310:K1=V1:K2=V:F=0:IFV+V1=0THEN
      ¬2230
1130 GOSUB2310:L1=V1:L2=V:LN=L1+(256*L2):
      ¬PRINTLN,
1150 GOSUB2330:IFV=32ORV=58THEN1150
1170 IFV=0THEN1200
1180 IFV<>143THEN1240
1190 GOSUB2330:IFV>0THEN1190
1200 F=1:FORX=0TON:IFTL(X)<LNTHENNEXTX
1220 IFTL(X)=LNTHEN1240
1230 GOTO1110
1240 PRINT#6,CHR$(K1);CHR$(K2);:PRINT#6,
      ¬CHR$(L1);CHR$(L2);:R=4:IFFTHENPRINT#
      ¬6,":";:R=5
1270 F=0:GOTO1360
1340 PRINT#6,C$;:R=R+1
1350 GOSUB2330
1360 IFV=137THENF=1
1370 IFV=139ORV=167THENF=1
1380 IFV=0THEN1820
1390 IFV=32THEN1350
1460 IFV<>143THEN1550
1470 GOSUB2330:IFV>0THEN1470
1480 GOTO1820
1550 IFV<>34THEN1690
```

```
 1560 PRINT#6,C$;:R=R+1:GOSUB2330:IFV=34THE
       ¬N1340
 1590 IFV>0THEN1560
 1600 IFFTHENV=0:GOTO1050
 1610 PRINT#6,CHR$(34);:R=R+1:GOTO1820
 1690 IFV<>58THEN1340
 1700 GOSUB2330:IFV=32ORV=58THEN1700
 1720 IFV=143THEN1470
 1730 IFV=0THEN1820
 1740 PRINT#6,":";:R=R+1:GOTO1360
 1820 IFFOR(R>170)THENV=0:GOTO1050
 1830 GOSUB2310:IFV+V1=0THEN2230
 1850 GOSUB2310:LN=V1+(256*V):L1=V1:L2=V:
       ¬PRINTLN,:FORX=0TON:IFTL(X)<LNTHENNEX
       ¬TX
 1940 IFTL(X)=LNTHEN2110
 2010 GOSUB2330:IFV=143THEN1470
 2020 IFV=32ORV=58THEN2010
 2030 IFV=0THEN1830
 2040 PRINT#6,":";:R=R+1:GOTO1360
 2110 PRINT#6,CHR$(0);CHR$(1);CHR$(1);:
       ¬PRINT#6,CHR$(L1);CHR$(L2);:R=4
 2130 GOSUB2330:IFV=32ORV=58THEN2130
 2150 IFV=0ORV=143THENPRINT#6,":";
 2160 F=0:GOTO1360
 2230 PRINT#6,CHR$(0);CHR$(0);CHR$(0);:
       ¬PRINT"ñ┌DONE↓↓":GOTO2430
 2310 GOSUB2330:V1=V
 2330 GET#5,C$:GOSUB2370:IFC$=""THENV=0:
       ¬RETURN
 2350 V=ASC(C$):RETURN
 2370 INPUT#15,EN,EM$,ET,ES:IFEN=0THENRETUR
       ¬N
 2400 PRINT:PRINT"↓↓↓┌DISK ERROR↓":
       ¬PRINTEN;EM$;ET;ES
 2430 CLOSE5:CLOSE6:CLOSE15
READY.
```

Feed Your PET Some APPLESOFT

G. A. Campbell

At first glance, this would seem to be an impossible task, but after reading this article, it makes sense. You can load and modify APPLE programs to run on your PET. Not impossible — that's incredible!

We all know that there is no such thing as compatibility in the world of personal computers. For example, the APPLE and the PET store programs on tape quite differently. However, by using the program in Listing 1, you can load programs from an APPLE directly into a PET. To be more specific, you can load APPLESOFT programs (cassette or ROM versions) into an upgrade-ROM PET. Conversion to original-ROM PET's is trivial.

Structure of an APPLESOFT Tape

One of the things which make the process fairly easy is the simple way that APPLEs SAVE programs. A bit is stored as one full cycle. on tape. A short cycle is a zero-bit, one about twice as long is a one-bit, and leader is slightly longer again. A byte is simply made up of eight bits, unlike the PET, which has a start-bit and a parity-bit. The high-order bit comes first.

A program is stored in two blocks. The first is a length block. It contains four bytes:

low-order half of program length
high-order half of program length
fixed hexadecimal "55"
checksum of the above.

The checksum is formed by beginning with hexadecimal "FF," then doing an exclusive-or on each byte of the block.

The second block contains the exact image of the program as it resides in memory. It is suffixed by two bytes, the second of which is a checksum formed the same way as for the length block. These two bytes are not counted in the program length.

Each block is preceded on tape by about ten seconds of leader (long bits) and one zero-bit, and followed by some tape which is effectively blank.

The other thing which makes the task easy is that both APPLESOFT and PET BASIC were written by Microsoft, and thus

programs have exactly the same format in memory.

The APPLESOFT Loader

The program in listing one contains many comments to point out the subtleties of how it operates. The major functions are:

Initialize everything upon entry so the program can be rerun if there is an error.

Time the cycles passing the head on the cassette.

Throw away the first bit.

Wait for the start-bit.

Makes bytes out of the following bits.

Do the checksum on the length block, and set up to read the actual program.

Convert the statement pointers if the program was cassette APPLESOFT.

Translate the BASIC tokens.

Convert the statement pointers from beginning at hexadecimal 0801 to hexadecimal 0501.

Move the program down from 0801 to 0501. (The code to do this is at the start of the program, since part of the loader is overlaid.)

Memory Requirements

The loader reads programs into the same location as ROM-based APPLESOFT. This is hexadecimal 0801, which is just above the screen on an APPLE. However, by the time the process is completed, the program has shuffled down to 0401. Thus, on an 8K PET you can load 6K of program text. Ignoring memory differences due to conversion, you have an additional 1K available for variables. APPLESOFT is also available as a loadable program (as opposed to ROM), in which case the APPLE requires 11K more than the PET to hold the same program.

Program Operation

The steps to load an APPLESOFT program are:

From BASIC, load the "APPLESOFT LOADER."

Type RUN, but don't press RETURN.

Position the APPLE tape at the beginning of the tone for the program you want. For the first program on a tape, just do a rewind. Otherwise, you will need an audio cassette-player. The

person who provided the APPLE tape should be able to show you how to position tape, since they do it all the time.

Press PLAY and wait three to nine seconds.

Press RETURN.

There are several possible results. The good one is that the PET displays OK and READY. Stop the tape and type 0 (zero) and RETURN. This deletes line zero, which is the last remnant of the loader, this is safe even if the APPLE program has a line zero, since only the first one is deleted. The APPLE program is now available for any required conversion. (See below)

About half the time, a question mark will print. This is followed by a BRK, which puts you in the machine language monitor on the upgrade-ROM PET. Type X to return to BASIC and try again. There was a checksum error on the length block. The error was possibly caused by the tape being positioned incorrectly. If you obtain the question mark a couple of times, try changing the 3E (decimal 62) which is stored at hexadecimal E811 by the routine named INIT to 3C (decimal 60). The APPLE is not consistent on whether a cycle is low-high or high-low. Since the loader only notices one transition per cycle, catching the wrong one gives it half of that bit, and half of this bit. Garble is the result. Fortunately, the program block always seems to be consistent with the length block. The time spent establishing which way to go is slight, since the length block ends about 11 seconds into the tape.

You may get the message TOO BIG immediately after reading the length block, which means the program won't fit into available memory.

The worst result is that the PET displays BAD. This means that there was a checksum error on the program block. It is necessary to reload the loader, and perhaps to reset the PET. I didn't see this result until I had succeeded in loading about 30 APPLE programs. The APPLE tends to like tapes which are a little quieter than PET tapes, so you might try getting a louder copy of the program.

Now the Fun Begins

Unfortunately, cassette tape format is not the only difference between the PET and the APPLE. After deleting line 0, you have a program loaded. You can list it, change it, or save it. But will it run? The answer is maybe. It can happen. But some programs will be hopeless. The APPLE has a very fancy graphics system, and APPLESOFT supports it. All the graphics commands are translated

into CMD (which the APPLE doesn't have). If there are any of these in the program you just loaded, you may have big trouble. Perhaps the person who gave you the APPLE tape can help you convert it, but it may not be worth the effort.

There are several other BASIC commands on the APPLE which are not available on the PET. The loader translates most of these into VERIFY, which is not supported by the APPLE. There are a few APPLE commands which are very easy to convert to the PET. The loader does phony translations on these. And finally, there are commands which translate exactly, but do not give the same result. The worst part of trying to correct these differences is that a line of BASIC can be 239 characters long on the APPLE, versus 80 on the PET. The longer lines will run just fine, but cannot easily be changed using the PET screen editor. Thus, you have to split this type of line into multiple lines.

The whole process will be greatly helped if you have an extended BASIC which includes the commands FIND and RENUMBER. This allows you to FIND commands which could cause problems, and split program lines without concern about smearing existing lines.

Space does not permit a complete tutorial on converting APPLESOFT programs. However, ignoring graphics, here are some suggestions:

Commands With No PET Equivalent

DEL - To delete program line. Unlikely to be imbedded in a program, since it also stops execution.
TRACE
NOTRACE - Usage obvious. Not needed in a working program.
POP - Cancel a GOSUB. This is an atrocious technique.
HIMEM - Set top of memory. Could be replaced with POKEs but is unlikely to be in a pure BASIC program which doesn't use graphics.
LOMEN - Set bottom of memory. Within a program it will probably cause the program to fail (even on the APPLE).
ONERR
RESUME - Replace with programmed editing.
SPEED= - Sets display rate. Replace with delay loops in key locations if necessary.
& - Does a jump to a machine language routine which the user must establish. Not part of normal BASIC programs.

NORMAL
FLASH
INVERSE -Adjusts the video mode for subsequent PRINT statements. The equivalent of INVERSE is specified within the text on the PET.

Commands With Phony Translation

TEXT/CONT - TEXT sets the text window to be the whole screen. CONT has no function within a program, so it is substituted. A program with multiple TEXT statements probably changes the size of the text window with POKE statements in order to print headings once, and then change what appears under them with PRINT statements.

HTAB/NEW - NEW has no function within a program except to make it commit suicide. HTAB is like TAB, but does not appear in a PRINT statement. HTAB can be directly converted to PRINTTAB(n-1); although it can very often be moved into an adjacent PRINT statement.

HOME/OPEN - OPEN is not supported by APPLESOFT. HOME clears the text window, so it can usually be replaced with PRINT"clr".

VTAB/CLOSE - CLOSE is not supported by APPLESOFT. VTABn positions the cursor on line n. Programs which use VTAB usually have lots of them, so at the start of the program define a string, for example DN$, containing a HOME character followed by 24 DOWNs. Then replace VTABn with PRINT LEFT$(DN$,n).

STORE/SAVE

RECALL/LOAD - It is assumed that you won't be converting programs with LOAD or SAVE in them. STORE and RECALL are used to dump matrices out to tape and read them back. Convert by putting in the appropriate OPEN, PRINT#, CLOSE or OPEN, INPUT#, CLOSE loops.

Commands Which May Give Different Results

PR#/PRINT# - Used to I-O to devices other than the screen and keyboard. Definitely not equivalent.

CALL/SYS - Used to invoke a machine language program. Almost certainly will require change. Note that CALL, WAIT, PEEK, and POKE on the APPLE may specify negative numbers. The address used will be 65536 minus the amount specified. This convention is a carryover from integer-BASIC, and has no equivalent on the PET. The most popular CALLs on the APPLE are:

-936 - clear the text window. Replace by printing a screen-clear.
-958 - clear the text window from the current print position. More difficult to replace.
-868 - clear from the current print position to the end of the line.

WAIT - Wait for an external event. Will require rework, since it references an actual memory location.
POKE - Sets a specific memory location to a particular value. Usually will require substantial rework.
PEEK - Returns the value stored in a specific memory location. Will also require rework.
USR - Another way to invoke a machine language routine.
RND - On the APPLE, RND(0) repeats the previous RND, unlike the PET, where it generates a truly random number.
GET - On the APPLE, this waits for a key to be pressed. On the PET, a null string is returned if no key has been pressed. To convert, make sure it is on a line by itself, and add a test like this:
nnnn GET A$: IF A$="" THEN nnnn
In the APPLE program there may be a PEEK at location -16384 to see if a key is being pressed which can be combined with the GET.
= - (Horrors. If you can't trust "=" what can you trust!) If the result of a comparison is used as a number, it will give a different result. For example, N=A=B sets N to a value depending on whether A equals B. On the APPLE, an equal condition gives a value of 1, on the PET, equal gives -1.
ASC Usually ASC of a letter is 64 greater on the APPLE than on the PET.
LIST - Terminates program execution on the PET, but not on the APPLE.
INPUT - APPLESOFT allows INPUT of a null string. You may encounter programs which invite you to "PRESS RETURN TO CONTINUE." On the PET, of course, you will obtain the READY. prompt and you are out of the program. Change the prompt to "PRESS A KEY TO CONTINUE," and replace the INPUT with a GET.
- INPUT generates a question mark prompt on the PET, but not on the APPLE.
BELL - On the APPLE, you can make the speaker beep by printing a control-G. No character appears on the screen. On the PET it prints as a reverse-G.
TAB - Use one position less on the PET.

PRINT - There are a number of detail differences. For example, tab-fields (invoked with commas) are 10 characters wide on the PET versus a sequence of 16,16,8,16,16,8. . . on the APPLE. A number is preceded by a space and followed by a skip on the PET, but not on the APPLE.

The Bottom Line

Does it work? It sure does! As long as you avoid graphics, you can have a program up and running in short order. I was able to load one Adventure-style game and have it completely running in less than half an hour. It sure beat keying in 16K of program text.

Many thanks must go to Keith Falkner of Toronto, who provided the description of what an APPLE tape looks like, many tapes to test with, and access to the manuals describing APPLESOFT.

```
                    0005              .LS
                    0010 ;
                    0020 ; APPLESOFT LOADER
                    0030 ;
                    0040 ; FOR USE ON THE COMMODORE PET/CBM
                    0050 ;
                    0060 ;    COPYRIGHT (C) 1980
                    0070 ;    GORD CAMPBELL
                    0080 ;    36 DOUBLETREE ROAD
                    0090 ;    WILLOWDALE, ONTARIO
                    0100 ;                  M2J 3Z4
                    0110 ;
                    0120 ; TO ASSEMBLE USING CARL MOSER'S
                    0130 ; ASSM/TED, REQUIRES 'SET' COMMAND
                    0140 ; AND A 32K MACHINE, SINCE THE SOURCE
                    0150 ; (INCLUDING COMMENTS) IS TOO LARGE
                           TO FIT
                    0160 ; INTO DEFAULT AREA, AND OBJECT
                    0170 ; GOES INTO THE DEFAULT TEXT AREA.
                    0180 ;
                    0190 WHERE      .DE 1
                    0200 ; USED FOR STORE INDIRECT
                    0210 ; THE ONLY PART OF PAGE ZERO
                    0220 ; WHICH IS SMEARED. IT DOESN'T
                    0230 ; MATTER, BECAUSE THE 'USR'
                    0240 ; VECTOR SHOULD BE SET UP BY
                    0250 ; ANY PROGRAM WHICH USES IT.
                    0260 PGMEN      .DE $2A
                    0270 ; BASIC 'END OF PROGRAM'
                    0280 ;
                    0290 ; CHANGE THIS TO $7C AND YOU
                    0300 ; ARE CONVERTED TO ORIGINAL ROM.
                    0310 ;
                    0320 PRINT      .DE $FFD2
                    0330 ; PRINT ROUTINE
                    0340            .BA $0400
                    0350            .OS
                    0360 ; HERE IS A BASIC PROGRAM.
0400- 00 0D 04      0370            .BY 0 $0D 4 0 0 $9E
0403- 00 00 9E
0406- 31 30 35      0380            .BY '1056:' $80
```

```
0409- 36 3A 80
040C- 00 00 00  0390            .BY 0 0 0 0
040F- 00
                0400 ;
                0410 ; IT READS '0 SYS1056:END'
                0420 ;
                0430 ;
                0440 ; -- VARIABLES --
                0450 ;
0410- 00 00 00  0460 LENGTH     .BY 00 00 00 00
0413- 00
                0470 ; APPLESOFT 'LENGTH' BLOCK
                0480 ; IS STORED HERE
0414- 00 00     0490 STLEN      .SI 0
                0500 ; LENGTH OF CURRENT BLOCK
0416- 00 00     0510 STLOC      .SI 0
                0520 ; WHERE IT GOES
0418- 00        0530 CHAR       .BY 0
                0540 ; CURRENT CHARACTER
0419- 00        0550 MODE       .BY 0
                0560 ; WHICH ACTIVITY NOW:
                0570 ;   0 - SYNCHRONIZING
                0580 ;   1 - LEADER
                0590 ;   2 - DATA
041A- 00        0600 BLOCK      .BY 0
                0610 ; WHICH BLOCK:
                0620 ;  0 - LENGTH BLOCK
                0630 ;  1 - PROGRAM BLOCK
041B- 42 41 44  0640 BAD        .BY 'BAD'
041E- 4F 4B     0650 OK         .BY 'OK'
                0660 ; CHECKSUM MESSAGES
                0670 ;
                0680 ; *** ENTRY POINT ***
                0690 ;
                0700 ;   MUST BE AT $0420
                0710 ;   FOR THE 'BASIC' PROGRAM
                0720 ;
0420- 4C 50 04  0730            JMP INIT
                0740 ; SKIP PAST CODE WHICH MOVES
                0750 ; THE PROGRAM DOWN FROM $0801
                0760 ; TO $0501. THIS CODE IS NEEDED
                0770 ; BECAUSE WHEN LINE ZERO (THE
                0780 ; PHONY BASIC PROGRAM) IS DELETED
                0790 ; 'END OF PROGRAM' ETC ARE ONLY
                0800 ; ADJUSTED BY ONE PAGE MAXIMUM.
                0810 ;
                0820 ; MOVE PROGRAM DOWN 3 PAGES
                0830 ;
0423- A9 08     0840 MOVE       LDA #8
0425- 85 02     0850            STA *WHERE+1
0427- A9 05     0860            LDA #5
0429- 85 2B     0870            STA *PGMEN+1
042B- A0 00     0880            LDY #0
042D- 84 01     0890            STY *WHERE
042F- 84 2A     0900            STY *PGMEN
0431- B1 01     0910 MOVLP      LDA (WHERE),Y
0433- 91 2A     0920            STA (PGMEN),Y
0435- E6 2A     0930            INC *PGMEN
0437- D0 02     0940            BNE MOVOK
0439- E6 2B     0950            INC *PGMEN+1
043B- A5 2A     0960 MOVOK      LDA *PGMEN
043D- C5 2C     0970            CMP *PGMEN+2
043F- D0 07     0980            BNE INWHERE
0441- A5 2B     0990            LDA *PGMEN+1
```

```
0443- C5 2D     1000            CMP *PGMEN+3
0445- D0 01     1010            BNE INWHERE
0447- 60        1020            RTS  ; FINISHED
0448- E6 01     1030 INWHERE    INC *WHERE
044A- D0 E5     1040            BNE MOVLP
044C- E6 02     1050            INC *WHERE+1
044E- D0 E1     1060            BNE MOVLP
                1070 ; ** ALWAYS GOES **
                1080 ;
                1090 ; INITIALIZATION
                1100 ;
                1110 ; SET UP POINTERS ETC ON ENTRY
                1120 ; SO IF WE HAD A BAD LOAD, WE
                1130 ; CAN TRY AGAIN BY ENTERING 'RUN'
                1140 ;
0450- A9 04     1150 INIT       LDA #4
0452- 8D 14 04  1160            STA STLEN
0455- 8D 17 04  1170            STA STLOC+1
0458- A9 10     1180            LDA #$10
045A- 8D 16 04  1190            STA STLOC
045D- A9 00     1200            LDA #0
045F- 8D 15 04  1210            STA STLEN+1
0462- 8D 19 04  1220            STA MODE
0465- 8D 1A 04  1230            STA BLOCK
0468- 78        1240            SEI
                1250 ; DISABLE INTERRUPTS
0469- AD 10 E8  1260            LDA $E810
                1270 ; CLEAR 6520
046C- A9 3E     1280            LDA #$3E
046E- 8D 11 E8  1290            STA $E811
                1300 ; MAKE 6520 RESPOND TO
                1310 ;  LOW TO HIGH TRANSITION
                1320 ;
                1330 ; FOR SOME TAPES THE '3E'
                1340 ; ABOVE MUST READ '3C'
                1350 ; (IE. HIGH TO LOW TRANSITION)
                1360 ;
0471- AD 16 04  1370            LDA STLOC
0474- 85 01     1380            STA *WHERE
0476- AD 17 04  1390            LDA STLOC+1
0479- 85 02     1400            STA *WHERE+1
                1410 ;
                1420 ; END OF INITIALIZATION
                1430 ;
047B- A0 08     1440 INITY      LDY #8
047D- A2 00     1450 INITX      LDX #0
047F- E8        1460 COUNT      INX
                1470 ; COUNT HOW MANY TIMES
                1480 ; THROUGH THE LOOP
0480- 2C 11 E8  1490            BIT $E811
                1500 ; HAVE WE A TRANSITION YET?
0483- 10 FA     1510            BPL COUNT
                1520 ; BRANCH BACK IF NOT YET
0485- AD 10 E8  1530            LDA $E810
                1540 ; RESET THE 6520
0488- AD 19 04  1550            LDA MODE
                1560 ; WHAT WERE WE DOING?
048B- F0 2C     1570            BEQ STARTUP
048D- C9 01     1580            CMP #1
048F- F0 2D     1590            BEQ STARTBIT
                1600 ; REAL DATA NOW
0491- E0 40     1610            CPX #$40
0493- 30 03     1620            BMI ZEROBIT
0495- 38        1630            SEC
0496- B0 01     1640            BCS SETBIT
```

```
                    1650 ; ** ALWAYS GOES **
0498- 18            1660 ZEROBIT    CLC
                    1670 ; THE CARRY BIT NOW INDICATES
                    1680 ; WHETHER WE GOT A ZERO OR ONE
0499- 2E 18 04      1690 SETBIT     ROL CHAR
                    1700 ; ROTATE IT INTO THE CHARACTER
049C- 88            1710            DEY
                    1720 ; FINISHED THIS CHARACTER?
049D- D0 DE         1730            BNE INITX    ; NO
049F- AD 18 04      1740            LDA CHAR
04A2- 91 01         1750            STA (WHERE),Y
                    1760 ; STORE THE CHARACTER
04A4- CE 14 04      1770            DEC STLEN
                    1780 ; REDUCE CHARACTER COUNT
04A7- D0 08         1790            BNE NEXTCHAR
04A9- AD 15 04      1800            LDA STLEN+1
                    1810 ; FINISHED THIS BLOCK?
04AC- F0 19         1820            BEQ FINMODE
04AE- CE 15 04      1830            DEC STLEN+1
04B1- E6 01         1840 NEXTCHAR   INC *WHERE
                    1850 ; INCREMENT DATA POINTER
04B3- D0 C6         1860            BNE INITY
04B5- E6 02         1870            INC *WHERE+1
04B7- D0 C2         1880            BNE INITY
                    1890 ;  ** ALWAYS GOES **
04B9- EE 19 04      1900 STARTUP    INC MODE
                    1910 ; THROW AWAY FIRST TRANSITION
04BC- D0 BF         1920            BNE INITX
                    1930 ;  ** ALWAYS GOES **
04BE- E0 40         1940 STARTBIT   CPX #$40
                    1950 ; IS IT A START BIT?
04C0- 10 BB         1960            BPL INITX    ; NO
04C2- EE 19 04      1970            INC MODE
04C5- D0 B6         1980            BNE INITX
                    1990 ; ** ALWAYS GOES **
04C7- AD 1A 04      2000 FINMODE    LDA BLOCK
                    2010 ; WE JUST LOADED A BLOCK.
                    2020 ; WHICH ONE WAS IT?
04CA- D0 62         2030            BNE LOADED
04CC- A9 FF         2040            LDA #$FF
04CE- 4D 10 04      2050            EOR LENGTH
                    2060 ; CHECKSUM ON LENGTH BLOCK
04D1- 4D 11 04      2070            EOR LENGTH+1
04D4- 4D 12 04      2080            EOR LENGTH+2
04D7- CD 13 04      2090            CMP LENGTH+3
04DA- F0 07         2100            BEQ NEXTBLK
04DC- A9 3F         2110            LDA #$3F
                    2120 ; BAD LOAD: PRINT QUESTION MARK
                    2130 ; AND QUIT WITH A 'BREAK'
04DE- 20 D2 FF      2140            JSR PRINT
04E1- 58            2150            CLI
                    2160 ; QUIT NOW
04E2- 00            2170            BRK
04E3- AD 10 04      2180 NEXTBLK    LDA LENGTH
                    2190 ; INITIALIZATION FOR PROGRAM LOAD
04E6- 8D 14 04      2200            STA STLEN
04E9- AD 11 04      2210            LDA LENGTH+1
04EC- 8D 15 04      2220            STA STLEN+1
04EF- EE 14 04      2230            INC STLEN
                    2240 ; LOAD CHECKSUM TOO
                    2250 ; MUST GO TWO BYTES PAST
                    2260 ; THE END OF THE ACTUAL PROGRAM
                    2270 ;
04F2- D0 03         2280            BNE LEN1
```

```
04F4- EE 15 04  2290              INC STLEN+1
04F7- EE 14 04  2300 LEN1         INC STLEN
04FA- D0 03     2310              BNE LENOK
04FC- EE 15 04  2320              INC STLEN+1
04FF- A9 08     2330 LENOK        LDA #$08
                2340 ;
                2350 ; ALWAYS LOAD AT $0801
                2360 ; IF IT'S CASSETTE APPLESOFT
                2370 ; CONVERT IT LATER
                2380 ;
0501- 85 02     2390              STA *WHERE+1
0503- A9 01     2400              LDA #$01
0505- 85 01     2410              STA *WHERE
0507- A5 35     2420              LDA *PGMEN+11
0509- CD 15 04  2430              CMP STLEN+1
050C- D0 13     2440              BNE DIFFPAGE
                2450 ; CHECKING ON WHETHER THERE IS
                2460 ; ENOUGH MEMORY.
050E- AD 14 04  2470              LDA STLEN
0511- C5 34     2480              CMP *PGMEN+10
0513- 90 0E     2490              BCC MEMOK
0515- A2 06     2500 MEMBAD       LDX #6
0517- BD 5F 06  2510 MEMCHR       LDA TOOBIG,X
051A- 20 D2 FF  2520              JSR PRINT
051D- CA        2530              DEX
051E- 10 F7     2540              BPL MEMCHR
                2550 ; MESSAGE DISPLAYED: QUIT NOW
0520- 60        2560              RTS
0521- 90 F2     2570 DIFFPAGE     BCC MEMBAD
0523- A9 00     2580 MEMOK        LDA #$00
0525- 8D 19 04  2590              STA MODE
0528- EE 1A 04  2600              INC BLOCK
052B- 4C 7B 04  2610              JMP INITY
052E- 58        2620 LOADED       CLI
                2630 ; ALLOW INTERRUPTS NOW
052F- A5 01     2640              LDA *WHERE
                2650 ;SET HIGH ADDRESS
0531- 8D 16 04  2660              STA STLOC
0534- A5 02     2670              LDA *WHERE+1
0536- 8D 17 04  2680              STA STLOC+1
                2690 ; INITIALIZATION FOR CHECKSUM
                2700 ; AND PROGRAM LINKAGE
                2710 ;
0539- A9 08     2720              LDA #8
053B- 85 02     2730              STA *WHERE+1
053D- A9 05     2740              LDA #5
053F- 8D 02 04  2750              STA $0402
0542- A9 01     2760              LDA #1
0544- 85 01     2770              STA *WHERE
0546- 8D 01 04  2780              STA $0401
0549- 8D 19 04  2790              STA MODE
                2800 ;NOW USE 'MODE' AS QUOTE-MODE FLAG
                2810 ; VALUES ARE:
                2820 ;   0 - CURRENTLY INSIDE QUOTES
                2830 ;   1 - NOT IN QUOTES
                2840 ;
054C- A9 FF     2850              LDA #$FF
054E- A0 00     2860              LDY #0
0550- 51 01     2870 CHKLOOP      EOR (WHERE),Y
                2880 ; CHECKSUM CALCULATION
0552- E6 01     2890              INC *WHERE
0554- D0 02     2900              BNE CHKEND
0556- E6 02     2910              INC *WHERE+1
0558- A6 01     2920 CHKEND       LDX *WHERE
```

```
055A- EC 16 04  2930            CPX STLOC
055D- D0 F1     2940            BNE CHKLOOP
055F- A6 02     2950            LDX *WHERE+1
0561- EC 17 04  2960            CPX STLOC+1
0564- D0 EA     2970            BNE CHKLOOP
0566- D1 01     2980            CMP (WHERE),Y
0568- F0 15     2990            BEQ CHKOK
056A- AD 1B 04  3000            LDA BAD
                3010 ; PRINT 'BAD'
056D- 20 D2 FF  3020            JSR PRINT
0570- AD 1C 04  3030            LDA BAD+1
0573- 20 D2 FF  3040            JSR PRINT
0576- AD 1D 04  3050            LDA BAD+2
0579- 20 D2 FF  3060            JSR PRINT
                3070 ; DO THE REST ANYWAY
057C- 4C 8B 05  3080            JMP CASSREL
057F- AD 1E 04  3090 CHKOK      LDA OK
                3100 ; PRINT 'OK'
0582- 20 D2 FF  3110            JSR PRINT
0585- AD 1F 04  3120            LDA OK+1
0588- 20 D2 FF  3130            JSR PRINT
058B- A9 01     3140 CASSREL    LDA #1
058D- 85 01     3150            STA *WHERE
058F- A9 08     3160            LDA #8
0591- 85 02     3170            STA *WHERE+1
0593- AD 02 08  3180            LDA $0802
0596- C9 08     3190            CMP #$08
0598- F0 16     3200            BEQ TRANS
059A- A0 01     3210 CASSLP     LDY #1
                3220 ; IT'S CASSETTE APPLESOFT
                3230 ; ORIGINAL ADDRESS WAS $3001
059C- B1 01     3240            LDA (WHERE),Y
059E- F0 EB     3250            BEQ CASSREL
                3260 ; ON THE SECOND PASS, IT LOOKS
                3270 ; LIKE ROM APPLESOFT
05A0- 38        3280            SEC
05A1- E9 28     3290            SBC #$28
05A3- 91 01     3300            STA (WHERE),Y
05A5- AA        3310            TAX
05A6- 88        3320            DEY
05A7- B1 01     3330            LDA (WHERE),Y
05A9- 85 01     3340            STA *WHERE
05AB- 86 02     3350            STX *WHERE+1
05AD- 4C 9A 05  3360            JMP CASSLP
05B0- A0 00     3370 TRANS      LDY #0
05B2- B1 01     3380            LDA (WHERE),Y
05B4- AA        3390            TAX
05B5- D0 08     3400            BNE NOTEN
05B7- C8        3410            INY
                3420 ; LAST LINE OF TOKENS DONE?
05B8- B1 01     3430            LDA (WHERE),Y
05BA- D0 03     3440            BNE NOTEN
05BC- 4C 12 06  3450            JMP TOKDONE
05BF- A0 01     3460 NOTEN      LDY #1
05C1- B1 01     3470            LDA (WHERE),Y
05C3- 8E 16 04  3480            STX STLOC
                3490 ; SET END OF CURRENT LINE
05C6- 8D 17 04  3500            STA STLOC+1
05C9- A0 04     3510            LDY #4
05CB- E6 01     3520 TOTXT      INC *WHERE
                3530 ; STEP PAST POINTER
                3540 ; AND LINE NUMBER
05CD- D0 02     3550            BNE WHOK
05CF- E6 02     3560            INC *WHERE+1
```

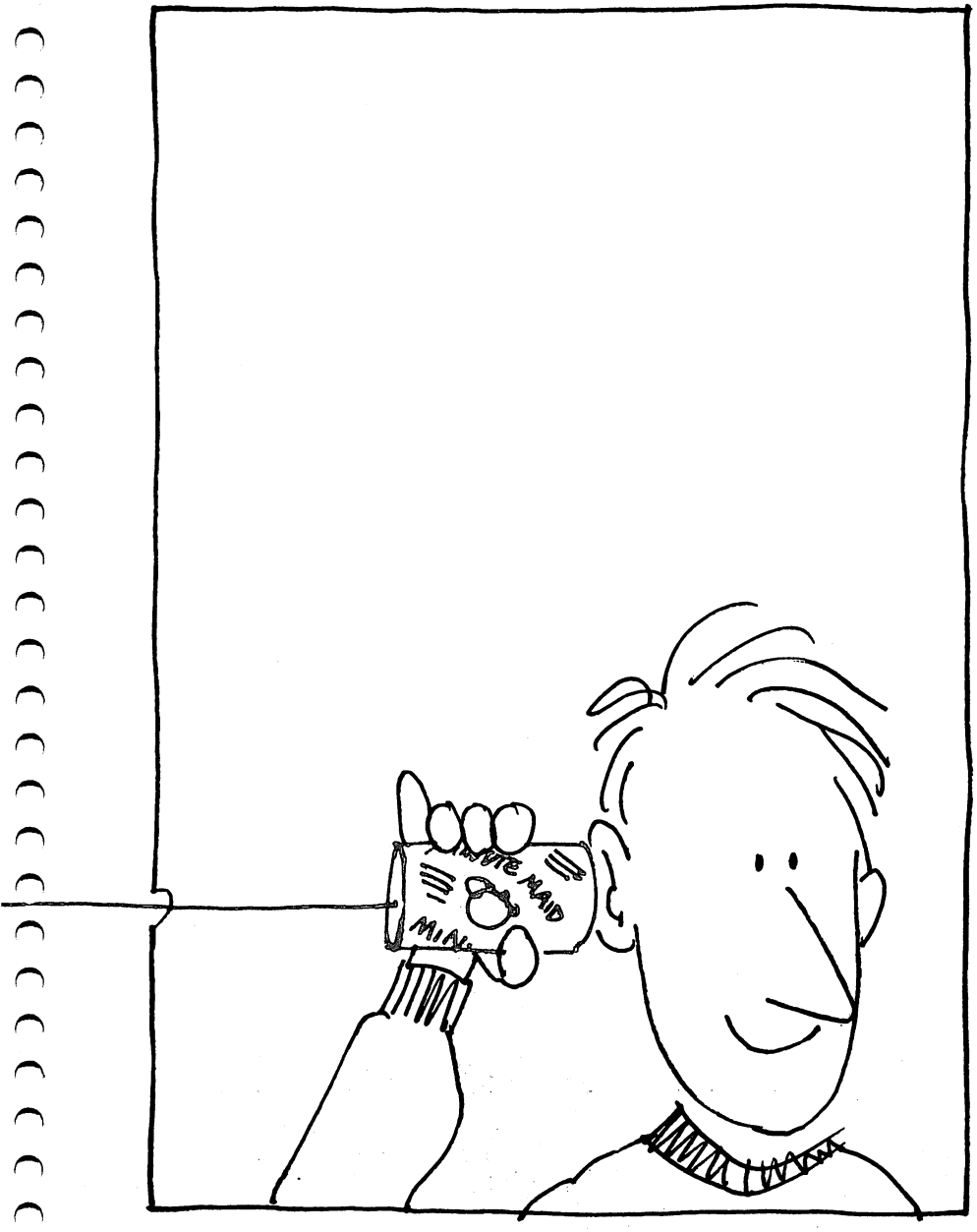
```
05D1- 88          3570 WHOK        DEY
05D2- D0 F7       3580             BNE TOTXT
05D4- B1 01       3590 TRLOOP      LDA (WHERE),Y
05D6- C9 22       3600             CMP #$22
                  3610 ; IS IT A QUOTE?
05D8- D0 0F       3620             BNE NOQ
05DA- AD 19 04    3630             LDA MODE
05DD- F0 05       3640             BEQ MODEON
05DF- CE 19 04    3650             DEC MODE
05E2- F0 12       3660             BEQ NXTCHAR
                  3670 ; ** ALWAYS GOES **
05E4- EE 19 04    3680 MODEON      INC MODE
05E7- D0 0D       3690             BNE NXTCHAR
                  3700 ; ** ALWAYS GOES **
05E9- AE 19 04    3710 NOQ         LDX MODE
05EC- F0 08       3720             BEQ NXTCHAR
                  3730 ; BRANCH IF WE ARE IN QUOTES
05EE- AA          3740             TAX
05EF- 10 05       3750             BPL NXTCHAR
                  3760 ; ONLY TRANSLATE TOKENS
05F1- BD 00 07    3770             LDA $0700,X
                  3780 ; TRANSLATE FROM TABLE
05F4- 91 01       3790             STA (WHERE),Y
05F6- E6 01       3800 NXTCHAR     INC *WHERE
05F8- D0 02       3810             BNE WHEOK
05FA- E6 02       3820             INC *WHERE+1
05FC- A5 01       3830 WHEOK       LDA *WHERE
05FE- CD 16 04    3840             CMP STLOC
                  3850 ; HAVE WE FINISHED THIS LINE?
0601- D0 D1       3860             BNE TRLOOP
0603- A5 02       3870             LDA *WHERE+1
0605- CD 17 04    3880             CMP STLOC+1
0608- D0 CA       3890             BNE TRLOOP
060A- A9 01       3900             LDA #1
060C- 8D 19 04    3910             STA MODE
                  3920 ; RESET QUOTE MODE FLAG
060F- 4C B0 05    3930             JMP TRANS
                  3940 ;
                  3950 ; FINISHED TOKEN TRANSLATION
0612- EE 16 04    3960 TOKDONE     INC STLOC
                  3970 ; INCLUDE THE '00 00' (END OF
                  3980 ; PROGRAM) IN THE LENGTH
                  3990 ;
0615- D0 03       4000             BNE MORLOC
0617- EE 17 04    4010             INC STLOC+1
061A- EE 16 04    4020 MORLOC      INC STLOC
061D- D0 03       4030             BNE LOCDON
061F- EE 17 04    4040             INC STLOC+1
0622- AD 16 04    4050 LOCDON      LDA STLOC
0625- 85 2C       4060             STA *PGMEN+2
0627- 85 2E       4070             STA *PGMEN+4
0629- AD 17 04    4080             LDA STLOC+1
062C- 38          4090             SEC
062D- E9 03       4100             SBC #$03
062F- 85 2D       4110             STA *PGMEN+3
0631- 85 2F       4120             STA *PGMEN+5
                  4130 ;
                  4140 ; SET UP PROGRAM LINKS FOR
                  4150 ; MOVE FROM $0801 TO $0501
                  4160 ;
0633- A9 01       4170             LDA #1
0635- 85 01       4180             STA *WHERE
0637- A9 08       4190             LDA #8
0639- 85 02       4200             STA *WHERE+1
```

```
063B- A0 00     4210 RELLP      LDY #0
063D- B1 01     4220            LDA (WHERE),Y
063F- 8D 14 04  4230            STA STLEN
0642- C8        4240            INY
0643- B1 01     4250            LDA (WHERE),Y
0645- F0 15     4260            BEQ RELDONE
0647- 8D 15 04  4270            STA STLEN+1
064A- 38        4280            SEC
064B- E9 03     4290            SBC #3
064D- 91 01     4300            STA (WHERE),Y
064F- AD 14 04  4310            LDA STLEN
0652- 85 01     4320            STA *WHERE
0654- AD 15 04  4330            LDA STLEN+1
0657- 85 02     4340            STA *WHERE+1
0659- 4C 3B 06  4350            JMP RELLP
065C- 4C 23 04  4360 RELDONE    JMP MOVE
065F- 47 49 42  4370 TOOBIG     .BY 'GIB OOT'
0662- 20 4F 4F
0665- 54
                4380 ; MESSAGE 'TOO BIG' REVERSED
                4390            .BA $0780
                4400 ;
                4410 ; **** TOKEN TRANSLATION TABLE ***
                4420 ;
                4430 ; (SEE FOOTNOTES BELOW)
                4440 ;
0780- 80        4450      .BY $80       ; END
0781- 81        4460      .BY $81       ; FOR
0782- 82        4470      .BY $82       ; NEXT
0783- 83        4480      .BY $83       ; DATA
0784- 85        4490      .BY $85       ; INPUT
0785- 95        4500      .BY $95       ; *DEL
0786- 86        4510      .BY $86       ; DIM
0787- 87        4520      .BY $87       ; READ
0788- 9D        4530      .BY $9D       ; *GR
0789- 9A        4540      .BY $9A       ; *TEXT/CONT **
078A- 98        4550      .BY $98       ; PR#/PRINT# *
078B- 84        4560      .BY $84       ; IN#/INPUT# *
078C- 9E        4570      .BY $9E       ; CALL/SYS   *
078D- 9D        4580      .BY $9D       ; *PLOT
078E- 9D        4590      .BY $9D       ; *HLIN
078F- 9D        4600      .BY $9D       ; *VLIN
0790- 9D        4610      .BY $9D       ; *HGR2
0791- 9D        4620      .BY $9D       ; *HGR
0792- 9D        4630      .BY $9D       ; *HCOLOR=
0793- 9D        4640      .BY $9D       ; *HPLOT
0794- 9D        4650      .BY $9D       ; *DRAW
0795- 9D        4660      .BY $9D       ; *XDRAW
0796- A3        4670      .BY $A3       ; *HTAB/NEW  **
0797- 9F        4680      .BY $9F       ; *HOME/OPEN **
0798- 9D        4690      .BY $9D       ; *ROT=
0799- 9D        4700      .BY $9D       ; *SCALE=
079A- 9D        4710      .BY $9D       ; *SHLOAD
079B- 95        4720      .BY $95       ; *TRACE
079C- 95        4730      .BY $95       ; *NOTRACE
079D- 95        4740      .BY $95       ; *NORMAL
079E- 95        4750      .BY $95       ; *INVERSE
079F- 95        4760      .BY $95       ; *FLASH
07A0- 9D        4770      .BY $9D       ; *COLOR=
07A1- 95        4780      .BY $95       ; *POP
07A2- A0        4790      .BY $A0       ; *VTAB/CLOSE **
07A3- 95        4800      .BY $95       ; *HIMEM
07A4- 95        4810      .BY $95       ; *LOMEM
07A5- 95        4820      .BY $95       ; *ONERR
```

```
07A6- 95        4830      .BY $95      ; *RESUME
07A7- 93        4840      .BY $93      ; *RECALL/LOAD **
07A8- 94        4850      .BY $94      ; *STORE/SAVE  **
07A9- 95        4860      .BY $95      ; *SPEED=
07AA- 88        4870      .BY $88      ; LET
07AB- 89        4880      .BY $89      ; GOTO
07AC- 8A        4890      .BY $8A      ; RUN
07AD- 8B        4900      .BY $8B      ; IF
07AE- 8C        4910      .BY $8C      ; RESTORE
07AF- 95        4920      .BY $95      ; *&
07B0- 8D        4930      .BY $8D      ; GOSUB
07B1- 8E        4940      .BY $8E      ; RETURN
07B2- 8F        4950    .BY $8F      ; REM
07B3- 90        4960    .BY $90      ; STOP
07B4- 91        4970    .BY $91      ; ON
07B5- 92        4980    .BY $92      ; WAIT *
07B6- 93        4990    .BY $93      ; LOAD
07B7- 94        5000    .BY $94      ; SAVE
07B8- 96        5010    .BY $96      ; DEF
07B9- 97        5020    .BY $97      ; POKE *
07BA- 99        5030    .BY $99      ; PRINT
07BB- 9A        5040    .BY $9A      ; CONT
07BC- 9B        5050    .BY $9B      ; LIST
07BD- 9C        5060    .BY $9C      ; CLEAR
07BE- A1        5070    .BY $A1      ; GET   *
07BF- A2        5080    .BY $A2      ; NEW
07C0- A3        5090    .BY $A3      ; TAB( *
07C1- A4        5100    .BY $A4      ; TO
07C2- A5        5110    .BY $A5      ; FN
07C3- A6        5120    .BY $A6      ; SPC(
07C4- A7        5130    .BY $A7      ; THEN
07C5- 9D        5140    .BY $9D      ; *AT
07C6- A8        5150    .BY $A8      ; NOT
07C7- A9        5160    .BY $A9      ; STEP
07C8- AA        5170    .BY $AA      ; +
07C9- AB        5180    .BY $AB      ; -
07CA- AC        5190    .BY $AC      ; * (TIMES)
07CB- AD        5200    .BY $AD      ; /
07CC- AE        5210    .BY $AE      ; ^ (EXPONENTIATION)
07CD- AF        5220    .BY $AF      ; AND
07CE- B0        5230    .BY $B0      ; OR
07CF- B1        5240    .BY $B1      ; >
07D0- B2        5250    .BY $B2      ; =
07D1- B3        5260    .BY $B3      ; <
07D2- B4        5270    .BY $B4      ; SGN
07D3- B5        5280    .BY $B5      ; INT
07D4- B6        5290    .BY $B6      ; ABS
07D5- B7        5300    .BY $B7      ; USR *
07D6- B8        5310    .BY $B8      ; FRE
07D7- 9D        5320    .BY $9D      ; *SCRN(
07D8- 9D        5330    .BY $9D      ; *PDL
07D9- B9        5340    .BY $B9      ; POS
07DA- BA        5350    .BY $BA      ; SQR
07DB- BB        5360    .BY $BB      ; RND *
07DC- BC        5370    .BY $BC      ; LOG
07DD- BD        5380    .BY $BD      ; EXP
07DE- BE        5390    .BY $BE      ; COS
07DF- BF        5400    .BY $BF      ; SIN
07E0- C0        5410    .BY $C0      ; TAN
07E1- C1        5420    .BY $C1      ; ATN
07E2- C2        5430    .BY $C2      ; PEEK *
07E3- C3        5440    .BY $C3      ; LEN
07E4- C4        5450    .BY $C4      ; STR$
07E5- C5        5460    .BY $C5      ; VAL
```

```
07E6- C6          5470    .BY $C6       ; ASC
07E7- C7          5480    .BY $C7       ; CHR$
07E8- C8          5490    .BY $C8       ; LEFT$
07E9- C9          5500    .BY $C9       ; RIGHT$
07EA- CA          5510    .BY $CA       ; MID$
                  5520 ;
                  5530 ; REMAINDER NOT IMPLEMENTED
                  5540 ; SUBSTITUTE 'REM'
                  5550 ;
07EB- 8F 8F 8F    5560    .BY $8F $8F $8F $8F $8F $8F
07EE- 8F 8F 8F
07F1- 8F 8F 8F    5570    .BY $8F $8F $8F $8F $8F $8F
07F4- 8F 8F 8F
07F7- 8F 8F 8F    5580    .BY $8F $8F $8F $8F $8F $8F
07FA- 8F 8F 8F
07FD- 8F 8F 8F    5590    .BY $8F $8F $8F
                  5600 ;
                  5610 ; COMMANDS WHICH ARE PRECEEDED BY
                  5620 ; AN ASTERISK ABOVE ARE NOT
                  5630 ; IMPLEMENTED ON THE PET. THE ONES
                  5640 ; WHICH DEPEND ON APPLE HARDWARE
                  5650 ; (GRAPHICS AND PDL) ARE TRANSLATED
                  5660 ; INTO 'CMD', THE OTHERS INTO
                  5670 ; 'VERIFY'
                  5680 ;
                  5690 ; COMMANDS WITH AN ASTERISK TO
                  5700 ; THE RIGHT MAY TRANSLATE BADLY.
                  5710 ;
                  5720 ; COMMANDS WITH TWO ASTERISKS ARE
                  5730 ; PHONY TRANSLATIONS FOR MANUAL
                  5740 ; CONVERSION.
                  5750 ;
                  5760 ; SEE ARTICLE FOR DETAILS.
                  5770 ;
                  5780 ; END OF PHONY BASIC PROGRAM
0800- 00 00 00    5790          .BY 0 0 0
                  5800          .EN
```

CHAPTER SIX: Communications



TelePET

Jim Butterfield

It's helpful to know just how telecommunication equipment works with the PET and how you can get the most out of it. Mr. Butterfield lucidly explains . . .

This is the age of computers talking to other computers. There's no reason why your PET can't join in the conversation, too. New communications interfaces for the PET are being announced fairly often these days. What's involved in the hookup?

Most commercial offerings give you the whole package to enable you to hook up and be "on the air" fairly quickly. But since there technical approaches are different, it's worthwhile to look at what a communications interface needs to do.

Interface Elements

There are several programs that need to be addressed in order to hook your PET to a telephone line. Starting at the telephone end, they are:

1. The telephone company gets annoyed if you wire things directly to the telephone line, unless they are "approved" devides. The small user should also worry about the dangers to his PET: some hefty voltages can come from the telephone exchange.

The easiest solution to this is an acoustic coupler. You fit your telephone handset into one of these, and it arranges to make noises into the transmitter and to listen to the earpiece with a microphone. No electrical connection — sound power does the whole job.

2. The telephone system was designed to carry voice, or sounds in a certain frequency range. The PET signal needs to be changed to an audible signal in order to be transmitted; at the other end, the sound frequencies need to be changed back into bits — the ones and zeros that the PET needs.

This program is solved by a device called a Modem. A Modem consists of two parts: a modulator, which changes bits to tone frequencies for sending; and a demodulator, which changes the tones back to bits.

3. You can normally send and/or receive only one bit at a time. PET handles eight bits at a time. Something has to take the eight bits from the PET (the "parallel" signal, since eight bits come out together) and fire them off one bit at a time (creating a "serial" signal, with one bit after the other). In the other direction, you must

collect the eight bits, one at a time, pack them together and deliver them to the PET as a parallel eight-bit byte.

Tied into this problem of parallel-to-serial conversion is a related job. Much of the time PET will have nothing to send. We must distinguish between an idle connection, where nothing is being sent, and an active connection which has a character under way.

This last task is usually effected by a signal called a *start bit*. The start bit is sent before the PET's information bits; it says, "here comes a character." If you don't use a start bit, you know that the line is idle.

All of the tasks above can be performed in machine language programs, or in a rather clever chip called a UART. Either way, you must arrange to send a start bit, then the eight data bits, one at a time, and then a brief pause (sometimes called a *stop bit*) before you start the next character. Coming the other way, the receiving PET must wait for a start bit and then collect the eight data bits into a single byte.

4. If you're communicating with a non-PET at the distant end, the other computer will probably want to receive a standard code called ASCII, and will send that code back to you. PET does not store characters in ASCII format, so that a little translation will be needed in both directions.

PET has characters that don't exist in ASCII. For example, most of the PET graphic characters don't have any corresponding ASCII characters. You'll have to give them up.

There are a few ASCII characters that don't have any counterpart in the PET. Most of the these are called *control* characters. You'll probably need a few of these for a good communications interface. Most commercial packages make them available with a two-key combination from the PET. For example, the keys Reverse, semicolon often generate the character known as ESC or Escape in ASCII; this character usually tells the distant computer to stop whatever it's doing and wait for a new command from you. It's a very handy character to know when the distant computer has started to send out a massive amount of data which you realize you really don't want.

5. The physical connection at the PET is either the IEEE-488 bus or the Parallel User Port. If it's the IEEE-488 bus, the connected device will have to obey the protocols — recognizing when it's selected, receiving and delivering characters to the bus, etc.

If it's the Parallel User Port, PET will need to contain a

machine language program which is called by the user's program any time it is desired to receive or send.

The IEEE-488 bus is simple to use — a normal PRINT # command will send data — but since the bus is shared with other devices, careful design is needed.

Tracing the Flow

Let's put the above together and track a character from the PET to the line, and vice-versa.

1. PET decides to send a character. If the interface is via the IEEE bus, PET might simply issue the command PRINT #7, "A"; or if the interface is via the parallel user port, the program might say, SYS 30456,"A". There are many possible variations.
2. The character — in this case, the letter A which is represented in PET text mode as hexadecimal C1 — must be translated to true ASCII. This might be done in either program or in hardware; in either case, the result is hexadecimal 41.
3. The parallel to serial translation now takes place. Once again, this may be done within a program or by hardware (a UART chip). A start bit is generated followed by the eight bits of data; each is sent at the appropriate time.
4. Each bit, as it is generated, is translated by the modem into an appropriate tone frequency. One frequency represents a zero bit, another represents a one bit.
5. The tones generated by the modem are fed into a small speaker which is very close to the telephone handset transmitter. The sound from the speaker is picked up by the telephone and sent to the line. It's on the way . . .

At the receiving end:

6. The telephone earpiece has been making a whining sound from the tone received from the line. The sound is picked up by a small microphone close to the earpiece.
7. The signal reaches the modem which examines the tone and classifies it as either a logic zero or a logic one. It passes along the logic state, zero or one, to the serial to parallel translator.
8. The serial to parallel translator waits patiently for a start bit (logic zero) to be received. When it sees this, it carefully collects the eight data bits at the appropriate times. This might be done either in a program or in hardware (again, with a UART).
9. The eight-bit character might be placed into a buffer or might just be held for pickup by the PET. In either case, the received character will need to be translated from ASCII into PET format.

The Modem/Acoustic Coupler

The modem and acoustic coupler are invariably packaged together. Speeds up to 30 characters per second are generally available; lower speeds will work, but the highest rate of 30 cps is a virtual standard now.

The Commodore interface packages everything into the modem/coupler case: IEEE bus interface, UART, the whole thing. Other suppliers use standard, commercially available modem/couplers and supply extra hardware and/or programs to complete the interface.

The commercially available modems use an interface known as RS-232. It's nice to have this interface available, since you connect other things besides modems to it. Various types of terminals, both video or hard copy, will hook up with no problems.

Parallel/Serial interfaces and Buffering

It's economical and flexible to use a program to do your parallel/serial interface, and buffering can be provided quite easily. It does take up memory space, however, and it can keep the PET rather busy; bits move in and out at a rate of one every three milliseconds or so. Your interface from BASIC will be rather ore tricky, too: PRINT # or GET # won't make the connection too easily.

Hardware costs more, but helps with some of these problems. You may not be liberated from the need for special programs, though. The mighty UART chip can only catch or send one character at a time. Unless you have buffering, PET will have to wait before the next character can be sent or received.

The GPIB bus

The IEEE-488 bus is ideal for sending or receiving characters from BASIC. As always, however, there's a catch or two. If the device you're sending to is busy and can't catch the character you want to send it, it will probably hang up the bus so that everything stops until it's ready. The same thing may happen if you try to INPUT or GET a character or value that hasn't arrived yet; you'll either time out or wait.

This isn't new. Many devices hold up the IEEE bus — the printer and the disk do it, for example. But with a communications interface, waiting time becomes a serious problem. You might lose a character if the bus is hung up waiting for something else to happen. It becomes more important to use the bus in a more sophisticated way.

Looking them Over

All of the above problems have been solved in a variety of ways by the various suppliers. A remarkable amount of ingenuity has been called into play, and the user has considerable choice.

Check out the units available to see which ones fit your style. How much of the package is hardware, and how much software? How easily can you interface with your own BASIC programs? Can you attach devices other than a modem? Does the unit contain buffering? How is the translation to and from ASCII accomplished? Can you abandon ASCII if you choose and send directly from PET to PET, for graphics or program transfer? How much memory will you need in the PET? Will you need disk? And, of course, how much money will it all cost?

There's no single answer. Find out what suits you.

Communications interfaces are here. You'll see more of them used in the PET community. One of these days, you'll be tempted to join the network.

Basic CBM 8010 Modem Routines

Jim Butterfield

Jim Butterfield presents a pair of amazing little programs for telecommunications — in BASIC!

The programs given on page 7 of the 8010 Modem Operator's Manual don't seem to do the job. In particular, the ASCII interface program often crashes; prints peculiar things if you are receiving parity characters; and drops line characters from time to time.

Here are a couple of replacement programs that should do the job better.

ASCII Interface

Set the modem switches to OR (Originate) and HD (Half Duplex). One exception: if you're working an "echoplex" type of system, the distant computer will repeat back everything you send; in this case, set the switch to FD (Full Duplex).

The program takes a few seconds to set up its translation arrays. You may start the program before telephone connection is established.

Special control characters can be set up, depending on your needs. Note, for example, that the delete character has been implemented in this program: PET's delete, decimal value 20, will be translated to ASCII backspace, decimal value 8, and vice versa; you can see the coding on line 210. You may implement your own to suit the needs of the computer or network. To enable Control-P, more accurately known as DLE (Data Link Escape) you might code: T(176) = 16. This would translate PET's shifted-zero character, a square-corner with bit value 176, to the ASCII DLE character, value 16.

PET-to-PET Interface

Both users should set their modem switches to HD (Half Duplex). One user should set OR (Originate), and the other AN (Answer); it doesn't matter which user sets what, so long as they are different. Communication is two-way in either case.

Cursor controls, reverse screen, and graphics features are supported. A user can clear both screens with the CLR key.

The biggest operational problem is making sure you don't both try to talk at the same time. There's no flashing cursor to prompt you. You'll soon get used to waiting for a pause from the other PET before sending your own stuff.

General Comments

The business part of these programs — lines 300-320 — are under severe time constraints. If you modify the programs, check carefully to make sure you don't start losing the occasional character incoming from the line.

These programs are quite simple; they convert your PET into a CRT terminal. That's not a cost-effective way to use a PET (terminals are cheaper) and eventually you should anticipate fitting more sophisticated programs which will allow you to send and receive programs and files.

For communications to an ASCII system:

```
100 REM 8010 INTERFACE  JIM BUTTERFIELD
110 REM FOR ASCII LINES
120 REMARK: SET SWITCH TO HD
200 DIM F(255),T(255)
210 FOR J = 32 TO 64 : T(J) = J : NEXT J : T(13) = 13 :
    T(20) = 8
220 FOR J = 65 TO 90 : K = J + 32 : T(J) = K : NEXT J
230 FOR J = 91 TO 95 : T(J) = J : NEXT J
240 FOR J = 193 TO 218 : K = J-128 : T(J) = K : NEXT J
250 REM ADD EXTRA FUNCTIONS HERE
260 FOR J = 0 TO 255 : K = T(J) : IF K THEN F(K) = J :
    F(K + 128) = J
270 NEXT J
280 POKE 1020,0 : POKE 59468,14
290 OPEN 5,5 : PRINT "ASCII I/O READY"
300 GET A$ : IF A$<>"" THEN PRINT#5,CHR$
    (T(ASC(A$)));
310 GET#5, A$ : IF ST = 0 AND A$<>"" THEN
    PRINT CHR$(F(ASC(A$)));
320 GOTO 300
```

For communications to another PET:

```
100 REM 8010 INTERFACE  JIM BUTTERFIELD
110 REM  FOR PET INTERCOMMUNICATION
120 REMARK: SET SWITCH TO HD
280 POKE 1020,0 : POKE 59468,14 if text mode desired
290 OPEN 5,5 : PRINT "PET I/O READY"
300 GET A$ : IF A$<>"" THEN PRINT#5,A$;
310 GET#5,A$ : IF ST = 0 THEN PRINT A$;
320 GOTO 300
```

APPENDIX

PET In Transition

Jim Butterfield

If you own the Upgrade ROM, you can use this memory map for your own programs for Original ROM PETs.

A transition issue of the Pet Gazette is very appropriate, because the PET itself is in transition. New products and new software are going to change the nature of the machine. Old hands at PET systems use will have to learn new tricks.

A lot of "old" software won't work on the new machines. Those chess and music playing programs, for example, can't make the transition in their present form. Many of the POKEs and PEEKs have shifted to new locations. SYS, USR and WAIT commands will need reworking.

The machines themselves have a few hardware changes. A new memory arrangement eliminates screen hash. The screen can no longer be blanked, so that certain special effects (explosions, etc.) are difficult to achieve. The character generator has changed, giving an unfamiliar reversal of upper and lower case. The memory expansion edge connector is physically different; it appears as if Commodore doesn't intend it to be user accessible any more. Instead, a "motherboard" architecture is hinted at; and empty ROM sockets suggest that new software may be forthcoming. An assembler? New languages? It's anybody's guess right now.

Further hardware changes are rumoured. Most of the ones I hear are associated with screen format changes (80 characters? Colour? Programmable characters?)

With all these changes, what should the PET owner do? Stay with his original machine? Retrofit with the new ROM chips? Buy the new model?

My recommendation is this: upgrade with new ROMs, or buy a new unit; but either way, take the plunge. You'll want the new model if you are strong on large keyboards, green screens, and/or ROM expansion capability; otherwise, stay with your existing machines, but fit the new ROM programs.

There's too much good stuff in the new software to hold back. The limit on array size is lifted; tape files behave correctly; the IEEE-488 bus works better; the built-in Machine Language Monitor is very valuable; you can now pull the computer out of a total crash without losing memory; and numerous little improvements have

been made.

Commodore may introduce more ROMs in the future. But I believe that they won't tinker with lower memory (locations 0 to 1023 decimal) to any great extent. So an upgrade which is made now should last for a while.

Commercial software houses will have to wrestle with the upgrade, of course. Buyers will have to closely examine programs on sale to make sure that they are compatible with their computer model. "AC/DC" programs, which will run on any existing ROM, will be a help (I understand that such a version of Microchess will soon be available). Eventually, I believe that the upgraded ROMs will become standard, and most software will be written for them; the original ROM will fade out of the picture.

Clubs, and newsletters like The Pet Gazette, will also need to cope with this transition. Programs and techniques will have to be carefully identified: which ROM set will they work on? Where possible, two versions will be desirable.

Eventually — hopefully — we'll all settle back into a standard machine. And then we can focus our attention fully on the main objective: making it do the jobs we want to do.

Memory locations for ROM upgrade on PET computers - Jim Butterfield, Toronto

0000-0002	0-2	USR Jump instruction
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	BASIC input buffer pointer; # subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	DATA scan flag; LIST quote flag; memory flag
000A	10	Subscript flag; FNx flag
000B	11	0=input; 64=get; 152=read
000C	12	ATN sign flag; comparison evaluation flag
000D	13	input flag; suppress output if negative
000E	14	current I/O device for prompt-suppress
0011-0012	17-18	BASIC integer address (for SYS, GOTO, etc.)
0013	19	Temporary string descriptor stack pointer
0014-0015	20-21	Last temporary string vector
0016-001E	22-30	Stack of descriptors for temporary strings
001F-0020	31-32	Pointer for number transfer
0021-0022	33-34	Misc. number pointer
0023-0027	35-39	Product staging area for multiplication
0028-0029	40-41	Pointer; Start-of-BASIC memory
002A-002B	42-43	Pointer: End-of-BASIC, Start-of-Variables
002C-002D	44-45	Pointer: End-of-Variables, Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: Bottom-of-strings (moving down)

0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit of BASIC Memory
0036-0037	54-55	Current BASIC line number
0038-0039	56-57	Previous BASIC line number
003A-003B	58-59	Pointer to BASIC statement (for CONT)
003C-003D	60-61	Line number, current DATA line
003E-003F	62-63	Pointer to current DATA item
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048	72	Y save register, new-operator save
004A	74	Comparison symbol accumulator
004B-004C	75-76	Misc. numeric work area
004D-0050	77-80	Work area; garbage yardstick
0051-0053	81-83	Jump vector for functions
0054-0058	84-88	Misc. numeric storage area
0059-005D	89-93	Misc. numeric storage area
005E-0063	94-99	Accumulator #1: E,M,M,M,M,S
0064	100	Series evaluation constant pointer
0065	101	Accumulator hi-order propogation word
0066-006B	102-107	Accumulator #2
006C	108	Sign comparison, primary vs. secondary
006D	109	low-order rounding byte for Acc#1
006E-006F	110-111	Cassette buffer length/Series pointer
0070-0087	112-135	Subrtn: Get BASIC Char; 77, 78=pointer
0088-008C	136-140	RND storage and work area
008D-008F	141-143	Jiffy clock for IT and TI$
0090-0091	144-145	Hardward interrupt vector
0092-0093	146-147	Break interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key depressed: 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant buffer
009D	157	Load=0, Verify=1
009E	158	# characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE-488 mode
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	PBD image for tape I/O
00A6	166	Key image
00A7	167	0=flashing cursor, else no cursor
00A8	168	Countdown for cursor timing
00A9	169	Character under cursor
00AA	170	Cursor blink flag
00AB	171	EOT bit received

00AC	172	Input from screen/input from keyboard
00AD	173	X save flag
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B4	180	Tape buffer character
00B5	181	Pointer in file name transfer
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Countdown for tape write
00BB	187	Tape buffer #1 count
00BC	188	Tape buffer #2 count
00BD	189	Write leader count; Read pass 1/pass 2
00BE	190	Write new byte; Read error flag
00BF	191	Write start bit; Read bit seq error
00C0	192	Pass 1 error log pointer
00C1	193	Pass 2 error correction pointer
00C2	194	0=Scan; 1-15=Count; $40=Load; $80=End
00C3	195	Checksum
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7-00C8	199-200	Utility pointer: tape buffer, scrolling
00C9-00CA	201-202	Tape end address/end of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	00=direct cursor, else programmed cursor
00CE	206	Timer 1 enabled for tape read; 00=disabled
00CF	207	EOT signal received from tape
00D0	208	Read character error
00D1	209	#characters in file name
00D2	210	Current logical file number
00D3	211	Current secondary addrs, or R/W command
00D4	212	Current device number
00D5	213	Line length (40 or 80) for screen
00D6-00D7	214-215	Start of tape buffer, address
00D8	216	Line where cursor lives
00D9	217	Last key input; buffer checksum; bit buffer
00DA-00DB	218-219	File name pointer
00DC	220	Number of keyboard INSERTs outstanding
00DD	221	Write shift word/Receive input character
00DE	222	#blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	Screen line table: hi order address & line wrap
00F9	249	Cassette #1 status switch
00FA	250	Cassette #2 status switch
00FB-00FC	251-252	Tape start address
0100-010A	256-266	Binary to ASCII conversion area
0100-013E	256-318	Tape read error log for correction
0100-01FF	256-511	Processor stack area

0200-0250	512-592	BASIC input buffer
0251-025A	593-602	Logical file number table
025B-0264	603-612	Device number table
0265-026E	613-622	Secondary address, or R/W cmd, table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape #1 buffer
033A-03F9	826-1017	Tape #2 buffer
03FA-03FB	1018-1019	Vector for Machine Language Monitor
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E0F8	49152-57592	Microsoft BASIC interpreter
E0F9-E7FF	57593-59391	Keyboard, Screen, Interrupt programs
E810-E813	59408-59411	PIA1-Keyboard I/O
E820-E823	59424-59427	PIA2-IEEE-488 I/O
E840-E84F	59456-59471	VIA-I/O and Timers
F000-FFFF	61440-65535	Reset, tape, diagnostic monitor

A Few Entry Points, Original/Upgrade 4.0 ROM

Jim Butterfield

Entry points seen in various programmers' machine language programs. The user is cautioned to check out the various routines carefully for proper setup before calling, registers used, etc.

ORIG	UPGR	4.0	DESCRIPTION
C357	C355	B3CD	?OUT OF MEMORY
C359	C357	B3CF	Send BASIC error message
C38B	C389	B3FF	Warm start, BASIC
C3AC	C3AB	B41F	Crunch & insert line
C430	C439	B4AD	Fix chaining & READY.
C433	C442	B4B6	Fix chaining
C48D	C495	B4FB	Crunch tokens
C522	C52C	B5A3	Find line in BASIC
C553	C55D	B5D4	Do NEW
C567	C572	B5E9	Reset BASIC and do CLR
C56A	C575	B5EC	Do CLR
C59A	C5A7	B622	Reset BASIC to start
C6B5	C6C4	B74A	Continue BASIC execution
C863	C873	B8F6	Get fixed-point number from BASIC.
C9CE	C9DE	BADB	Send Return, LF if in screen mode
C9D2	C9E2	BADF	Send Return, Linefeed
CA27	CA1C	BB1D	Print string
CA2D	CA22	BB23	Print precomputed string
CA47	CA43	BB44	Print "?"
CA49	CA45	BB46	Print character
CE11	CDF8	BEF5	Check for comma
CE13	CDFA	BEF7	Check for specific character
CE1C	CE03	BF00	'SYNTAX ERROR'
CFD7	CFC9	C187	Find fl-pt variable, given name
D079	D069	C2B9	Bump Variable Address by 2
D0A7	D09A	C2EA	Float to Fixed conversion
D278	D26D	C4BC	Fixed to Float conversion
D679	D67B	C8D7	Get byte to X reg
D68D	D68F	C8EB	Evaluate String
D6C4	D6C6	C921	Get two parameters
D73C	D773	C99D	Add (from memory)
D8FD	D934	CB5E	Multiply by memory location
D9B4	D9EE	CC18	Multiply by ten

DA74	DAAE	CCD8	Unpack memory variable to Accum#1
DAA9	DAE3	CD0D	Copy Acc #1 to (X,Y) location
DB1B	DB55	CD7F	Completion of Fixed to Float conversion
DC9F	DCD9	CF83	Print fixed-point value
DCA9	DCE3	CF8D	Print floating-point value
DCAF	DCE9	CF93	Convert number to ASCII string
E3EA	E3D8	E202	Print a character
na	E775	D722	Output byte as 2 hex digits
na	E7A7	D754	Input 2 hex digits to A
na	E7B6	D763	Input 1 hex digit to A
E7DE	F156	F185	Print system message
F0B6	F0B6	F0D2	Send 'talk' to IEEE
F0BA	F0BA	F0D5	Send 'listen' to IEEE
F12C	F128	F143	Send Secondary Address
E7DE	F156	F185	Send canned message
F167	F16F	F19E	Send character to IEEE
F17A	F17F	F1B6	Send 'untalk'
F17E	F183	F1B9	Send 'unlisten'
F187	F18C	F1C0	Input from IEEE
F2C8	F2A9	F2DD	Close logical file
F2CD	F2AE	F2E2	Close logical file in A
F32A	F301	F335	Check for Stop key
F33F	F315	F349	Send message if Direct mode
na	F322	F356	LOAD subroutine
F3DB	F3E6	F425	?LOAD ERROR
F3E5	F3EF	F42E	Print READY & reset BASIC to start
F3FF	F40A	F449	Print SEARCHING. . .
F411	F41D	F45C	Print file name
F43F	F447	F486	Get LOAD/SAVE type parameters
F462	F466	F4A5	Open IEEE channel for output.
F495	F494	F4D3	Find specific tape header block
F504	F4FD	F53C	Get string
F52A	F521	F560	Open logical file from input parameters
F52D	F524	F563	Open logical file
F579	F56E	F5AD	?FILE NOT FOUND, clear I/O
F57B	F570	F5AF	Send error message
F5AE	F5A6	F5E5	Find any tape header block
F64D	F63C	F67B	Get pointers for tape LOAD
F667	F656	F695	Set tape buffer start address
F67D	F66C	F6AB	Set cassette buffer pointers
F6E6	F6F0	F72F	Close IEEE channel
F78B	F770	F7AF	Set input device from logical file number
F7DC	F7BC	F7DF	Set output device from LFN.
F83B	F812	F857	PRESS PLAY..; wait
F85E	F835	F87A	Sense tape switch
F87F	F855	F89A	Read tape to buffer
F88A	F85E	F8A3	Read tape
F8B9	F886	F8CB	Write tape from buffer
F8C1	F88E	F8D3	Write tape, leader length in A
F913	F8E6	F92B	Wait for I/O complete or Stop key
FBDC	FB76	FBBB	Reset tape I/O pointer

FD1B	FC9B	FCE0	Set interrupt vector
FFC6	FFC6	FFC6	Set input device
FFC9	FFC9	FFC9	Set output device
FFCC	FFCC	FFCC	Restore default I/O devices
FFCF	FFCF	FFCF	Input character
FFD2	FFD2	FFD2	Output character
FFE4	FFE4	FFE4	Get character

Basic 4.0 Memory Map

Jim Butterfield

There are some differences in usage between the 40- and 80-column machines.

Hex	Decimal	Description
0000-0002	0-2	USR jump
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Input buffer pointer; # of subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	Flag: DATA scan; LIST quote; memory
000A	10	Subscript flag; FNX flag
000B	11	0=INPUT; $40=GET; $98=READ
000C	12	ATN sign/Comparison Evaluation flag
000D-000F	13-15	Disk status DS$ descriptor
0010	16	Current I/O device for prompt-suppress
0011-0012	17-18	Integer value (for SYS, GOTO etc)
0013-0015	19-21	Pointers for descriptor stack
0016-001E	22-30	Descriptor stack (temp strings)
001F-0022	31-34	Utility pointer area
0023-0027	35-39	Product area for multiplication
0028-0029	40-41	Pointer: Start-of-BASIC
002A-002B	42-43	Pointer: Start-of-Variables
002C-002D	44-45	Pointer: Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: String-storage (moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit-of-memory
0036-0037	54-55	Current BASIC line number
0038-0039	56-57	Previous BASIC line number
003A-003B	58-59	Pointer: BASIC statement for CONT
003C-003D	60-61	Current DATA line number
003E-003F	62-63	Current DATA address
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048-0049	72-73	Y-save; op-save; BASIC pointer save
004A	74	Comparison symbol accumulator
004B-0050	75-80	Misc work area, pointers, etc
0051-0053	81-83	Jump vector for functions
0054-005D	84-93	Misc numeric work area
005E	94	Accum#1: Exponent

005F-0062	95-98	Accum#1: Mantissa
0063	99	Accum#1: Sign
0064	100	Series evaluation constant pointer
0065	101	Accum#1 hi-order (overflow)
0066-006B	102-107	Accum#2: Exponent, etc.
006C	108	Sign comparison, Acc#1 vs #2
006D	109	Accum#1 lo-order (rounding)
006E-006F	110-111	Cassette buff len/Series pointer
0070-0087	112-135	CHRGET subroutine; get BASIC char
0077-0078	119-120	BASIC pointer (within subrtn)
0088-008C	136-140	Random number seed.
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	BRK interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key down; 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant for tape
009D	157	Load=0, Verify=1
009E	158	Number of characters in keybd buffer
009F	159	Screen reverse flag
00A0	160	IEEE output; 255=character pending
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE output buffer
00A6	166	Key image
00A7	167	0=flash cursor
00A8	168	Cursor timing countdown
00A9	169	Character under cursor
00AA	170	Cursor in blink phase
00AB	171	EOT received from tape
00AC	172	Input from screen/from keyboard
00AD	173	X save
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B3	179	Logical Address temporary save
00B4	180	Tape buffer character; MLM command
00B5	181	File name pointer; MLM flag, counter
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Tape writer countdown
00BB-00BC	187-188	Tape buffer pointers, #1 and #2
00BD	189	Write leader count; read pass½
00BE	190	Write new byte; read error flag
00BF	191	Write start bit; read bit seq error

00C0-00C1	192-193	Error log pointers, pass½
00C2	194	0=Scan/1-15=Count/$40=Load/$80=End
00C3	195	Write leader length; read checksum
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7-00C8	199-200	Utility pointer: tape, scroll
00C9-00CA	201-202	Tape end addrs/End of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	0=direct cursor, else programmed
00CE	206	Tape read timer 1 enabled
00CF	207	EOT received from tape
00DO	208	Read character error
00D1	209	# characters in file name
00D2	210	Current file logical address
00D3	211	Current file secondary addrs
00D4	212	Current file device number
00D5	213	Right-hand window or line margin
00D6-00D7	214-215	Pointer: Start of tape buffer
00D8	216	Line where cursor lives
00D9	217	Last key/checksum/misc.
00DA-00DB	218-219	File name pointer
00DC	220	Number of INSERTs outstanding
00DD	221	Write shift word/read character in
00DE	222	Tape blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	(40-column) Screen lien wrap table
00E0-00E1	224-225	(80-column) Top, bottom of window
00E2	226	(80-column) Left window margin
00E3	227	(80-column) Limit of keybd buffer
00E4	228	(80-column) Key repeat flag
00E5	229	(80-column) Repeat countdown
00E6	230	(80-column) New key marker
00E7	231	(80-column) Chime time
00E8	232	(80-column) HOME count
00E9-00EA	233-234	(80-column) Input vector
00EB-00EC	235-236	(80-column) Output vector
00F9-00FA	249-250	Cassette status, #1 and #2
00FB-00FC	251-252	MLM pointer/Tape start address
00FD-00FE	253-254	MLM, DOS pointer, misc.
0100-010A	256-266	STR$ work area, MLM work
0100-013E	256-318	Tape read error log
0100-01FF	256-511	Processor stack
0200-0250	512-592	MLM work area; Input buffer
0251-025A	593-602	File logical address table
025B-0264	603-612	File device number table
0265-026E	613-622	File secondary adds table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape #1 input buffer
033A-03F9	826-1017	Tape #2 input buffer
033A	826	DOS character pointer
033B	827	DOS drive 1 flag

033C	828	DOS drive 2 flag
033D	829	DOS length/write flag
033E	830	DOS syntax flags
033F-0340	831-832	DOS disk ID
0341	833	DOS command string count
0342-0352	834-850	DOS file name buffer
0353-0380	851-896	DOS command string buffer
03EE-03F7	1006-1015	(80-column) Tab stop table
03FA-03FB	1018-1019	Monitor extension vector
03FC	1020	IEEE timeout defeat
0400-7FFF	1024-32767	Available RAM including expansion
8000-83FF	32768-33791	(40-column) Video RAM
8000-87FF	32768-34815	(80-column) Video RAM
9000-AFFF	36864-45055	Available ROM expansion area
B000-DFFF	45056-57343	BASIC, DOS, Machine Lang Monitor
E000-E7FF	57344-59391	Screen, Keyboard, Interrupt programs
E810-E813	59408-59411	PIA 1 - Keyboard I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O and timers
E880-E881	59520-59521	(80-column) CRT Controller
F000-FFFF	61440-65535	Reset, I/O handlers, Tape routines

PET 4.0 ROM Routines

Jim Butterfield

The 40-character and 80-character machines are the same except for addresses $E000-$E7FF.

This map shows where various routines lie. The first address is not necessarily the proper entry point for the routine. Similarly, many routines require register setup or data preparation before calling.

	Description
B000-B065	Action addresses for primary keywords
B066-B093	Action addresses for functions
B094-B0B1	Hierarchy and action addresses for operators
B0B2-B20C	Table of BASIC keywords
B20D-B321	BASIC messages, mostly error messages
B322-B34F	Search the stack for FOR or GOSUB activity
B350-B392	Open up space in memory
B393-B39F	Test: stack too deep?
B3A0-B3CC	Check available memory
B3CD	Send canned error message, then:
B3FF-B41E	Warm start; wait for BASIC command
B41F-B4B5	Handle new BASIC line input
B4B6-B4E1	Rebuild chaining of BASIC lines
B4E2-B4FA	Receive line from keyboard
B4FB-B5A2	Crunch keywords into BASIC tokens
B5A3-B5D1	Search BASIC for given line number
B5D2	Perform NEW, and;
B5EC-B621	Perform CLR
B622-B62F	Reset BASIC execution to start
B630-B6DD	Perform LIST
B6DE-B784	Perform FOR
B785-B7B6	Execute BASIC statement
B7B7-B7C5	Perform RESTORE
B7C6-B7ED	Perform STOP or END
B7EE-B807	Perform CONT
B808-B812	Perform RUN
B813-B82F	Perform GOSUB
B830-B85C	Perform GOTO
B85D	Perform RETURN, then:
B883-B890	Perform DATA: skip statement
B891	Scan for next BASIC statement
B894-B8B2	Scan for next BASIC line
B8B3	Perform IF, and perhaps:
B8C6-B8D5	Perform REM: skip line
B8D6-B8F5	Perform ON
B8F6-B92F	Accept fixed-point number

B930-BA87	Perform LET
BA88-BA8D	Perform PRINT#
BA8E-BAA1	Perform CMD
BAA2-BB1C	Perform PRINT
BB1D-BB39	Print string from memory
BB3A-BB4B	Print single format character
BB4C-BB79	Handle bad input data
BB7A-BBA3	Perform GET
BBA4-BBBD	Perform INPUT#
BBBE-BBF4	Perform INPUT
BBF5-BC01	Prompt and receive input
BC02-BCF6	Perform READ
BCF7-BD18	Canned Input error messages
BD19-BD71	Perform NEXT
BD72-BD97	Check type mismatch
BD98	Evaluate expression
BEE9	Evaluate expression within parentheses
BEEF	Check parenthesis, comma
BF00-BF0B	Syntax error exit
BF8C-C046	Variable name setup
C047-C085	Set up function references
C086-C0B5	Perform OR, AND
C0B6-C11D	Perform comparisons
C11E-C12A	Perform DIM
C12B-C1BF	Search for variable
C1C0-C2C7	Create new variable
C2C8-C2D8	Setup array pointer
C2D9-C2DC	32768 in floating binary
C2DD-C2FB	Evaluate integer expression
C2FC-C4A7	Find or make array
C4A8	Perform FRE, and:
C4BC-C4C8	Convert fixed-to-floating
C4C9-C4CE	Perform POS
C4CF-C4DB	Check not Direct
C4DC-C509	Perform DEF
C50A-C51C	Check FNx syntax
C51D-C58D	Evaluate FNx
C58E-C59D	Perform STR$
C59E-C5AF	Do string vector
C5B0-C61C	Scan, set up string
C61D-C669	Allocate space for string
C66A-C74E	Garbage collection
C74F-C78B	Concatenate
C78C-C7B4	Store string
C7B5-C810	Discard unwanted string
C811-C821	Clean descriptor stack
C822-C835	Perform CHR$
C836-C861	Perform LEFT$
C862-C86C	Perform RIGHT$
C86D-C896	Perform MID$
C897-C8B1	Pull string data

C8B2-C8B7	Perform LEN
C8B8-C8C0	Switch string to numeric
C8C1-C8D0	Perform ASC
C8D1-C8E2	Get byte parameter
C8E3-C920	Perform VAL
C921-C92C	Get two parameters for POKE or WAIT
C92D-C942	Convert floating-to-fixed
C943-C959	Perform PEEK
C95A-C962	Perform POKE
C963-C97E	Perform WAIT
C97F-C985	Add 0.5
C986	Perform subtraction
C998-CA7C	Perform addition
CA7D-CAB3	Complement accum#1
CAB4-CAB8	Overflow exit
CAB9-CAF1	Multiply-a-byte
CAF2-CB1F	Constants
CB20	Perform LOG
CB5E-CBC1	Perform multiplication
CBC2-CBEC	Unpack memory into accum#2
CBED-CC09	Test & adjust accumulators
CC0A-CC17	Handle overflow and underflow
CC18-CC2E	Multiply by 10
CC2F-CC33	10 in floating memory
CC34	Divide by 10
CC3D	Perform divide-by
CC45-CCD7	Perform divide-into
CCD8-CCFC	Unpack memory into accum#1
CCFD-CD31	Pack accum#1 into memory
CD32-CD41	Move accum#2 to #1
CD42-CD50	Move accum#1 to #2
CD51-CD60	Round accum#1
CD61-CD6E	Get accum#1 sign
CD6F-CD8D	Perform SGN
CD8E-CD90	Perform ABS
CD91-CDD0	Compare accum#1 to memory
CDD1-CE01	Floating-to-fixed
CE02-CE28	Perform INT
CE29-CEB3	Convert string to floating-point
CEB4-CEE8	Get new ASCII digit
CEE9-CEF9	Constants
CF78	Print IN, then:
CF7F-CF92	Print BASIC line #
CF93-D0C6	Convert floating-point to ASCII
D0C7-D107	Constants
D108	Perform SQR
D112	Perform power function
D14B-D155	Perform negation
D156-D183	Constants
D184-D1D6	Perform EXP
D1D7-D220	Series evaluation

D221-D228	RND constants
D229-D281	Perform RND
D282	Perform COS
D289-D2D1	Perform SIN
D2D2-D2FD	Perform TAN
D2FE-D32B	Constants
D32C-D35B	Perform ATN
D35C-D398	Constants
D399-D3B5	CHRGET sub for zero page
D3B6-D471	BASIC cold start
D472-D716	Machine Language Monitor
D717-D7AB	MLM subroutines
D7AC-D802	Perform RECORD
D803-D837	Disk parameter checks
D838-D872	Dummy disk control messages
D873-D919	Perform CATALOG or DIRECTORY
D91A-D92E	Output
D92F-D941	Find spare secondary address
D942-D976	Perform DOPEN
D977-D990	Perform APPEND
D991-D9D1	Get disk status
D9D2-DA06	Perform HEADER
DA07-DA30	Perform DCLOSE
DA31-DA64	Set up disk record
DA65-DA7D	Perform COLLECT
DA7E-DAA6	Perform BACKUP
DAA7-DAC6	Perform COPY
DAC7-DAD3	Perform CONCAT
DAD4-DB0C	Insert command string values
DB0D-DB39	Perform DSAVE
DB3A-DB65	Perform DLOAD
DB66-DB98	Perform SCRATCH
DB99-DB9D	Check Direct command
DB9E-DBD6	Query ARE YOU SURE?
DBD7-DBE0	Print BAD DISK
DBE1-DBF9	Clear DS$ and ST
DBFA-DC67	Assembly disk command string
DC68-DE29	Parse BASIC DOS command
DE2C-DE48	Get Device Number
DE49-DE86	Get file name
DE87-DE9C	Get small variable parameter
** Entry points only for E000-E7FF **	
E000	Register/screen initialization
E0A7	Input from keyboard
E116	Input from screen
E202	Output character
E442	Main Interrupt entry
E455	Interrupt: clock, cursor, keyboard
E600	Exit from Interrupt
**	**
F000-F0D1	File messages

F0D2	Send 'Talk'
F0D5	Send 'Listen'
F0D7	Send IEEE command character
F109-F142	Send byte to IEEE
F143-F150	Send byte and clear ATN
F151-F16B	Option: timeout or wait
F16C-F16F	DEVICE NOT PRESENT
F170-F184	Timeout on read, clear control lines
F185-F192	Send canned file message
F193-F19D	Send byte, clear control lines
F19E-F1AD	Send normal (deferred) IEEE char
F1AE-F1BF	Drop IEEE device
F1C0-F204	Input byte from IEEE
F205-F214	Get a byte
F215-F265	INPUT a byte
F266-F2A1	Output a byte
F2A2	Abort files
F2A6-F2C0	Restore default I/O devices
F2C1-F2DC	Find/setup file data
F2DD-F334	Perform CLOSE
F335-F342	Test STOP key
F343-F348	Action STOP key
F349-F350	Send message if Direct mode
F351-F355	Test if Direct mode
F356-F400	Program load subroutine
F401-F448	Perform LOAD
F449-F46C	Print SEARCHING
F46D-F47C	Print LOADING or VERIFYING
F47D-F4A4	Get Load/Save parameters
F4A5-F4D2	Send name to IEEE
F4D3-F4F5	Find specific tape header
F4F6-F50C	Perform VERIFY
F50D-F55F	Get Open/Close parameters
F560-F5E4	Perform OPEN
F5E5-F618	Find any tape header
F619-F67A	Write tape header
F67B-F694	Get start/end addrs from header
F695-F6AA	Set buffer address
F6AB-F6C2	Set buffer start & end addrs
F6C3-F6CB	Perform SYS
F6CC-F6DC	Set tape write start & end
F6DD-F767	Perform SAVE
F768-F7AE	Update clock
F7AF-F7FD	Connect input device
F7FE-F84A	Connect output device
F84B-F856	Bump tape buffer pointer
F857-F879	Wait for PLAY
F87A-F88B	Test cassette switch
F88C-F899	Wait for RECORD
F89A	Initiate tape read
F8CB	Initiate tape write

F8E0-F92A	Common tape I/O
F92B-F934	Test I/O complete
F935-F944	Test STOP key
F945-F975	Tape bit timing adjust
F976-FA9B	Read tape bits
FA9C-FBBA	Read tape characters
FBBB-FBC3	Reset tape read address
FBC4-FBC8	Flag error into ST
FBC9-FBD7	Reset counters for new byte
FBD8-FBF3	Write a bit to tape
FBF4-FC85	Tape write
FC86-FCBF	Write tape leader
FCC0-FCDA	Terminate tape; restore interrupt
FCDB-FCEA	Set interrupt vector
FCEB-FCF8	Turn off tape motor
FCF9-FD0A	Checksum calculation
FD0B-FD15	Advance load/save pointer
FD16-FD4B	Power-on Reset
FD4C-FD5C	Table of interrupt vectors
** Jump table:	**
FF93-FF9E	CONCAT,DOPEN,DCLOSE,RECORD
FF9F-FFAA	HEADER,COLLECT,BACKUP,COPY
FFAB-FFB6	APPEND, DSAVE,DLOAD,CATALOG
FFB7-FFBC	RENAME,SCRATCH
FFBD	Get disk status
FFC0	OPEN
FFC3	CLOSE
FFC6	Set input device
FFC9	Set output device
FFCC	Restore default I/O devices
FFCF	INPUT a byte
FFD2	Output a byte
FFD5	LOAD
FFD8	SAVE
FFDB	VERIFY
FFDE	SYS
FFE1	Test stop key
FFE4	GET byte
FFE7	Abort all files
FFEA	Update clock
FFFA-FFFF	Hard vectors: NMI, Reset, INT

Index

Notes

Notes

COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868

In NC call 919-275-9809

Quantity	Title	Price	Total
_____	The Beginner's Guide to Buying A Personal Computer	$ 3.95**	_____
_____	COMPUTE!'s First Book of Atari	$12.95*	_____
_____	Inside Atari DOS	$19.95*	_____
_____	COMPUTE!'s First Book of PET/CBM	$12.95*	_____
_____	Programming the PET/CBM	$24.95***	_____
_____	Every Kid's First Book of Robots and Computers	$ 4.95**	_____
_____	COMPUTE!'s Second Book of Atari	$12.95*	_____
_____	COMPUTE!'s First Book of VIC	$12.95*	_____
_____	COMPUTE!'s First Book of Atari Graphics	$12.95*	_____
_____	Mapping the Atari	$14.95*	_____
_____	Home Energy Applications On Your Personal Computer	$14.95*	_____
_____	Machine Language for Beginners	$12.95*	_____

* Add $2 shipping and handling. Outside US add $4 air mail; $2 surface mail.

** Add $1 shipping and handling. Outside US add $4 air mail; $2 surface mail.

*** Add $3 shipping and handling. Outside US add $9 air mail; $3 surface mail.

Please add shipping and handling for each book ordered. _____

Total enclosed or to be charged. _____

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

☐ Payment enclosed Please charge my: ☐ VISA ☐ MasterCard
☐ American Express Acc't. No. Expires ___/___

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.

01-9