Exchange

October 1985

INSIDE: Pull-out BASIC 3.10 Reference Chart

Exchange of IBM PC Information

IBM

Exchange of IBM PC Information is a monthly publication of the National Distribution Division, International Business Machines Corporation, Boca Raton, Florida, USA.

Editor	Michael Engelberg
User Group Editor	Bernard Penney
Associate Editor, Design Director	Karen Porterfield
Writer	John Warnock
Editorial Assistants	Steve Mahlum
	Wayne Taylor
Illustrators	Michael Bartalos
	Jeff Jamison
	Narda Lebo
	John Segal
Production	Cohen and Company
User Group Support Manager	Gene Barlow

Exchange of IBM PC Information is distributed at no charge to registered PC user groups. To register with us, please write to:

IBM PC User Group Support
IBM Corporation (2900)
P.O. Box 3022
Boca Raton, FL 33431-0922

To correspond with *Exchange,* please write to:

Editor, *Exchange*
IBM Corporation (2900)
P.O. Box 3022
Boca Raton, FL 33431-0922

POSTMASTER: send address changes to *Exchange of IBM PC Information,* IBM Corporation (2900), P.O. Box 3022, Boca Raton FL 33431-0922.

IBM cannot be responsible for the security of material considered by other firms to be of a confidential or proprietary nature. Such information should not be made available to IBM.

IBM has tested the programs contained in this publication. However, IBM does not guarantee that the programs contain no errors.

IBM hereby disclaims all warranties as to materials and workmanship, either expressed or implied including without limitation, any implied warranty of merchantability or fitness for a particular purpose. In no event will IBM be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damage arising out of the use or inability to use any information provided through this service even if IBM has been advised of the possibility of such damages, or for any claim by any other party.

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

It is possible that the material in this publication may contain reference to, or information about, IBM products, programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming or services in your country.

The IBM Personal Computer XT/370

D.J. Meyer
University of Rochester
IBM PC User Group

As part of IBM's Academic Information System (ACIS), the University of Rochester often receives IBM equipment to examine. I work with one of the university's IBM mainframe computers, the IBM 4341 Model 2 running CMS. I also work with IBM PCs, so when I heard we were receiving an XT/370 to examine, I offered to take a look at it. In simple terms, an XT/370 is a PC equipped with a 10MB hard disk and two boards that run the 370 instruction set in the CMS operating environment. CMS is an operating environment typically found on large mainframe computers. The XT/370 also comes with the communication capabilities to talk with a CMS mainframe. The price of an XT/370 is at most one-tenth that of a CMS mainframe.

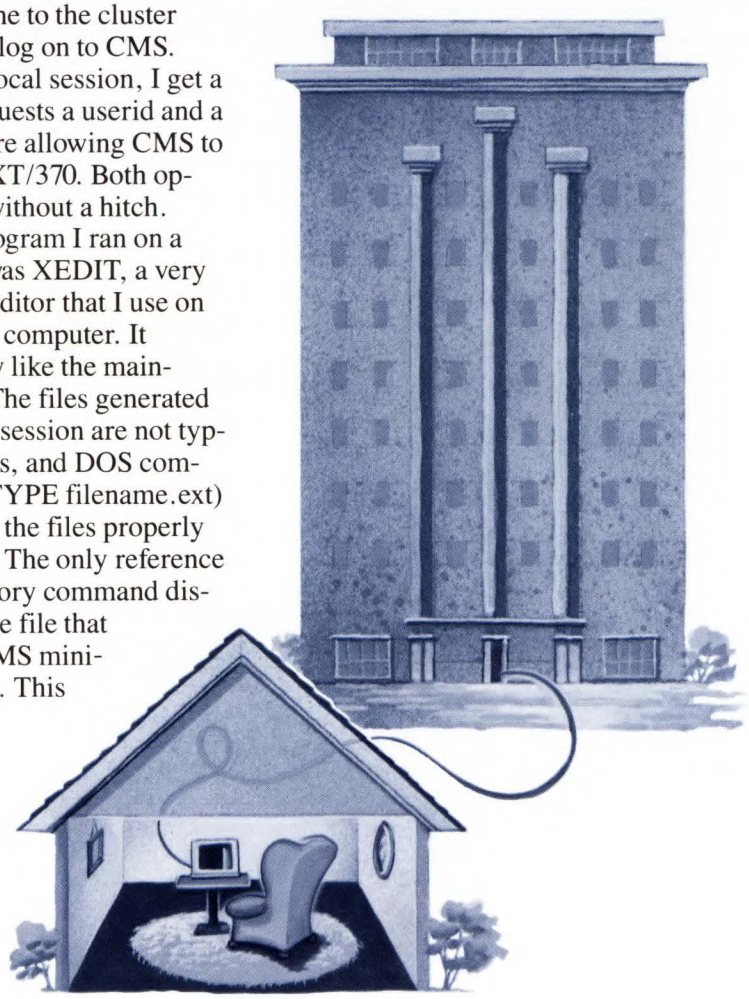
To get the XT/370 up and running, the following preparations were made. First, a coaxial line was dropped into my office to connect the XT/370 to the cluster controller, which in turn connects to the CMS mainframe. The coaxial line allows the XT/370 to communicate to the host. I also needed the Virtual Machine PC program (VMPC), and installed the CMS files that come on seven disks. This set up the system so that it could run DOS or VMPC.

To run CMS, I needed to install myself as a valid user on the XT/370. The XT/370 will support several separate users (not logged on simultaneously) if desired. Each user's files are protected with user identification and passwords, much like those used on the host CMS machine. I also needed to give myself a minidisk, which is the CMS concept of disk storage. Once this was done, I could tell the XT/370 to run VMPC.

The XT/370 gives the option of running a local or host 3270 session. When I run as a host, I can talk over the line to the cluster controller and log on to CMS. When I run a local session, I get a screen that requests a userid and a password before allowing CMS to be run on the XT/370. Both options worked without a hitch.

The first program I ran on a local session was XEDIT, a very powerful text editor that I use on the mainframe computer. It worked exactly like the mainframe editor. The files generated during a CMS session are not typical ASCII files, and DOS commands (e.g., TYPE filename.ext) cannot display the files properly on the display. The only reference a DOS DIRectory command displays is a single file that looks like a CMS minidisk definition. This file contains all the files on that specific CMS minidisk.

To transfer files between the host and local machine, I needed to install the host server programs on the mainframe. The installation was smooth and uneventful. To use the server programs to transfer files between the host and the XT/370, I had to log on to the mainframe and start the server, VMPCSERV, and toggle back to the local session. With the server active, files can be shared between the host and local machines as if they were ordinary, local CMS minidisks.



The XT/370 comes with no real programming languages other than REXX, so I tried moving a language onto it. REXX (Restructured EXtended eXecutor) is generally used as an interpreted command language and is much like Pascal in that one can write interpreted programs with it. I wanted a language I could compile, and decided on VS FORTRAN. (We first checked with IBM to be sure that copying VS FORTRAN to the XT/370 was legal.) Next, I issued a command to the local XT/370 of the format:

```
LINK userid itsvaddr
     myvaddr mode
     password
```

When the XT/370 realized that I wanted to link to a disk that it did not have, it checked the host. It found the host minidisk and linked to it. Next I entered:

```
ACCESS myvaddr fm
```

which told the XT/370 to access the disk. When I did a Query DISK, sure enough, there were all the PC minidisks and a host minidisk! I simply copied the VS FORTRAN module to the XT/370 with the command:

```
COPY FORTRAN
     MODULE fm=B.
```

After about forty minutes, I had all the files needed to run VS FORTRAN (about 1MB of files) right on top of my desk!

This brings up a key point. The VS FORTRAN module, or any other module, is just that—an executable image to the machine. It is **not** source code. That is the beauty of the XT/370. It runs the identical 370 instruction set, so executables generally can be transferred and run with no problems. I won't be so bold as to say absolutely no problems, since there are areas which VMPC does not support, and areas in which it would fail if a module should try to access it. For example, the PUNCH command is unsupported in VMPC, so if a module tried to PUNCH something, it wouldn't work. You can be assured, though, that it won't fail due to the 370 instruction set. I tested a small FORTRAN program that calculates prime numbers, and it compiled, loaded, and ran without a problem. I could just as easily have generated a module and transferred it back to the mainframe and executed it there.

In conclusion, I liked the machine. I thought it was wonderful to be able to run a local session of CMS and share minidisks between the host and local machines.

What might be a stronger point, though, would be to develop an application program on the mainframe, generate the module and transfer it to the XT/370, so that the application could be run elsewhere. I found that the compilation times left a lot to be desired. It was relatively slow. When I reminded myself of exactly what I was doing — running CMS and compiling VS FORTRAN on my desk — the slowness didn't bother me as much.

I have tried moving other relatively large modules to the XT/370 and found that they ran identically to the mainframe, but at a slower rate.

In conclusion, I liked the machine. I thought it was wonderful to be able to run a local session of CMS and share minidisks between the host and local machines, as if they were just different minidisks with a different file mode. I think it is an advantage to be able to use the same editor, programming languages, application programs and operating environment in both the local and host context. The XT/370 eases the difficulty of working between your home and your office, and frees you from having to contend with communications unless it's absolutely necessary. For example, you could write a program on the XT/370 anywhere, and later transfer it to a mainframe to execute the program.

My only reservation with the machine is that it is rather slow. It takes noticeable amounts of time to read and write from the disk, as well as to run programs. Much of this may be due to paging required when the virtual machine needs more than the physical 640KB on the XT/370. I think that the speed problem may be eased with an AT/370. The AT can address more than the 640K memory the XT is limited to. It also has a quicker processor on the motherboard, the 80286.

Overall, the concept is terrific: having some of the power of a mainframe in a compact box, at an affordable price. The only thing lacking is the responsiveness that we have come to expect from a big machine.

What's New in Personal Editor II

Jim Wyllie
IBM Corporation

Editor's note: This article, written by the program's author, contains detailed information about the changes and new features of Personal Editor II. It assumes you are familiar with the original IBM Personal Editor.

Personal Editor II, now available through *The Directory of Personally Developed Software,* contains a number of enhancements and improvements to Personal Editor. In this article, the two versions of the editor will be referred to as "PE1" and "PE2".

Improved Command Parsing and Key Definitions

PE2 retains almost no distinction between what PE1 called "commands" and "functions". In PE1, commands could be entered only by typing them on the command line or in a macro file; functions could be used only by assigning them to keys. PE2 removes both of these restrictions. You may assign any command to a key with the define command (as in "def a-f1 = [? MEMORY]" or "def 3 = [c/30/XXX/*]"). You may also type function names directly on the command line, e.g., [REFLOW] or [CENTER IN MARGINS].

By eliminating the function/command distinction, you can now use [EDIT] to switch among active files. In PE1, function key F8 cycles through the active files by writing the EDIT command on the command line and executing it, a procedure which erased the previous contents of the command line. Using [EDIT] allows you to cycle through the active files while preserving the contents of the command line and current cursor position.

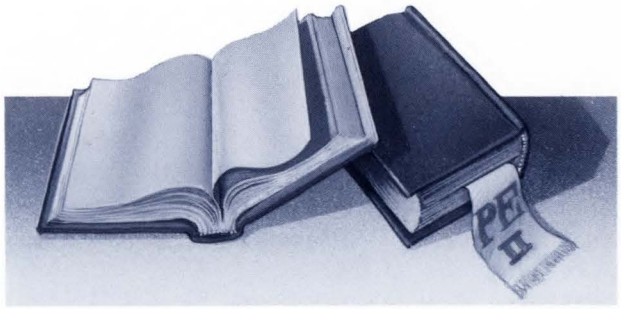
PE2 allows one key definition to invoke another. For example, the definition of key a-f1 could use the current definition of key a-f2 as follows:

```
def a-f1 = 'before' [KEY A-F2] 'after'
```

If the definition of a-f2 is

```
def a-f2 = ' then '
```

pressing a-f1 would yield the character string "before then after".



PE2 guards against the possibility of an infinite loop by limiting the levels of macro definitions to five. A loop would be executed five times before the message "Macros nested too deeply" would appear.

Included in PE2 is a set of pseudo keys intended for use only within complex macro definitions. PE1 uses unshifted, alphabetic shift (s-), control shift (c-), and alternate shift (a-) in combination with each of the 76 definable keys on the keyboard; PE2 also allows a user shift, indicated by the prefix u-. For example, if two key definitions share a long common expression, the common part could be assigned to the key u-f1 and used from within the main key definitions using the command [KEY U-F1]. The user-shift keys also can be invoked directly from the command line by typing [KEY U-...].

A toggling technique is also possible using chained key definitions:

```
def u-f1 = [DEF ENTER = [INSERT LINE]]
      [DEF A-i = [KEY U-F2]]
def u-f2 = [DEF ENTER = [DOWN]
      [BEGIN LINE]] [DEF A-i =
      [KEY U-F1]]
def a-i = [KEY U-F1]
```

The first time a-i is typed, the Enter key will be defined to insert lines. The next time a-i is typed, the Enter key will be redefined so that it just moves the cursor down to the beginning of the next line. Typing a-i will toggle between these two settings.

As the example above illustrates, commands (typically DEFINE commands) can be nested inside other commands by enclosing them in the proper number of square brackets ([ ]). For simple commands, the brackets can often be omitted. The DIR command, for example, can be typed with or without brackets.

All PE1 commands you could type on the command line can be entered without brackets in PE2. In addition, many of the commands that required brackets in PE1 (functions) are properly recognized by

PE2 without brackets. The exceptions are those commands that the PE2 parser could mistake for the LOCATE or CHANGE commands ("center line" looks like a CHANGE command using the letter "e" as the string delimiter, but [CENTER LINE] is properly recognized).

Also, PE2 allows two-character abbreviations for most of its commands. These abbreviations must always appear within square brackets. For example, [PD] may be used instead of PAGE DOWN. The new SET ABBREV command controls whether command abbreviations or full command names are generated in response to ? KEY or EDIT .KEYDEFS.

In PE2, you may assign definitions to 238 typeable keys, plus 76 user-shifted keys.

In PE2, the keys you can define using the DEFINE command include all keys and key combinations that can be distinguished by the standard BIOS keyboard handler, including the Ctrl-shifted alphabetic keys and the numeric keypad, plus the user-shifted keys described above. This means you may assign definitions to 238 typeable keys, plus 76 user-shifted keys. PE1 allows you to define only 99 keys. The alphabetic and numeric keys can be redefined in PE2 so that, for example, letters from foreign alphabets can be made more readily accessible. You might construct a PE2 profile that defines the alphabetic keys according to the Dvorak layout. The EDIT .KEYDEFS command shows a complete list of definable keys.

Alternate Ways to Enter Characters

To allow easy input of any of the 256 possible character codes, PE2 allows you to represent a character by its decimal number enclosed in square brackets. All three digits must be specified. For example, [000] represents the null character. Decimal character representation may be used as a component in any command; thus, the definitions

```
def a-f10 = 'abc'
```

and

```
def a-f10 = [097] [098] [099]
```

are equivalent. Since any command may be entered from the command line, you may place a character into a file by typing [NNN] on the command line, positioning the cursor in the proper place in the data area of a window, then using the [EXECUTE] command (c-enter by default) to "type" the character in the cursor position.

Arbitrary characters may also be entered by holding down the Alt key while typing the character number on the numeric keypad. Unlike PE1, PE2 does not require characters entered in this way to be preceded by [ESCAPE].

The query command corresponding to [NNN] is "? CHAR". This command places the bracketed decimal representation of the character at the cursor onto the command line. Try typing "? CHAR" on the command line, then move the cursor to a character in the data area, then type c-enter ([EXECUTE]).

Macro definitions can be much more complex in PE2 than in PE1. To assist you in finding errors in macro definitions, PE2 places the cursor at the point of an error in a key definition. For example, the definition

```
def a-f10 = 'hello"
```

is in error, since there is no single quote closing the string hello. PE2 produces the error message "Missing quote or]" in response to this definition, and places the cursor on the opening quote of the string.

Split Screen Support

PE2 lets you split the screen of its display into as many as four windows. Each window contains a section of any file. You may use windows to view several files at once, or you may display different portions of a single file in different windows. Either way, each window always contains an up-to-date image of the file section it contains. For instance, if the same file section is displayed in two windows, both windows will be updated every time a character is typed into either window. Similarly, if a marked area in one window is moved to another window, the data in the marked area will simultaneously disappear from one window and reappear in the other. Each window has its own independent cursor position, so you may freely move from one window to another window that contains a view of a different section of the same file without losing track of where you are in either place.

There are several new commands that control the PE2 split screen facility. When PE2 starts, the screen is treated as one large window, and appears exactly as it does in PE1. The [SPLIT SCREEN] command cycles through four possible screen organizations.

The first time you use [SPLIT SCREEN], PE2 splits the screen vertically into two windows, each 39 characters wide, separated by a double vertical line (one screen column is not used). Each window contains all the fields you would find in a full size window, including data area, command line, status line, and space for messages. Since the two windows are each narrower than the full screen, the names of the files displayed in each window may not completely fit on the status line, in which case PE2 will display "..." followed by the end of the file name. Executing [SPLIT SCREEN] a second time divides the screen into four windows. As before, each window behaves independently. The third [SPLIT SCREEN] divides the screen into two horizontal windows. The fourth [SPLIT SCREEN] returns the screen to a single window. The [SPLIT SCREEN] command is assigned to the c-s key by default.

Only one window can be active at any time. The active window contains the blinking cursor and displays the line and column number of the cursor position as well as the status of Insert/Replace mode. To move the cursor from one window to the next, use the [NEXT WINDOW] command (c-w by default). This command cycles clockwise through the windows.

Another command, [NEXT VIEW], is similar to [NEXT WINDOW], but [NEXT VIEW] will switch to the next window (in clockwise order) that displays the same file as the current window. The default key for [NEXT VIEW] is c-v. The [ZOOM WINDOW] command returns the screen to a single, large window containing the file that was active when [ZOOM WINDOW] was executed.

As part of the changes required for split-screen support, the algorithm for screen updates was substantially revised. PE2 defers all changes to the screen until all commands in a macro or a key definition have finished executing. When executing very long macros, this can be disconcerting, because nothing immediately happens on the display. However, PE2 will quickly finish the macro and update the screen. Unless you consecutively execute long macros, you will not be hampered by the pause. The advantage of the deferred update algorithm is that it allows much better response to operations such as word wrap while the same file is displayed in all four windows.

When running PE2 with the IBM Enhanced Graphics Adapter, fonts other than the standard 9-by-14 (IBM Monochrome Display), 8-by-8 (IBM Color Display), or 8-by-14 (IBM Enhanced Color Display) may be used. The new SET DISPLAY LARGE/SMALL FONT command may be used to select either the 8-by-8 or 8-by-14 fonts built into the Enhanced Graphics Adapter. For high resolution displays (Monochrome or ECD) this command selects between 25- and 43-line displays. For lower resolution displays (IBM Color Display), the command selects either a 14- or 25-line display.

DOS Path Support

PE2 allows full absolute or relative DOS path names (including drive letter) to be used in all commands involving file names. Either forward or reverse slash (/ or \) may be used as the path separator character (see page D-14 of the DOS 2.00 manual); however, PE2 always displays path names using reverse slash as the separator. PE2 parses relative path names according to the current directory and always displays full path names in the file name field on the status line. For example, if the current directory is \level 1\level 2 and the command "e ../new" is given, the file name that PE2 will display is "\level 1\new".

There are new commands to set and query the current working directory. The CHDIR (or CD) command operates as in DOS, except that the new default drive may also be specified. For example, if the current drive is C: and the current directory is \PE, the command "CD A:\newdir" would change the current drive to A and the current directory on that drive to \newdir. Complementing CHDIR is a new option of the Question-Mark command that queries the current directory. After the CD command in the example above is issued, the command "? dir" would produce "chdir a:\newdir" on the command line. As in DOS, the CHDIR command with no parameters displays the current directory; it is identical to the "? dir" command.

PE2 let you split the screen of its display into as many as four windows. Each window contains a section of any file.

The DIR, ERASE, and RENAME commands have been updated to use DOS path support. They can operate on files in any directory. In addition, the .dir file displayed in response to the DIR command now shows directories as well as files. However, wild-card characters are not supported in the ERASE command, so the command "ERASE *.pas" gives the message "File not found".

Command Line Parameters

PE2 accepts several options on the DOS command line when it is invoked. The command line consists of a sequence of tokens, separated from one another by one or more blanks. No blanks may appear within any token. Tokens beginning with forward slash (/) are the new command line options. The letter after the slash specifies which command line option is being set. The order of command options on the command line is not significant. If a command line option appears more than once, PE2 uses the last (rightmost) value. The last command line token not beginning with slash, if any, is used as the initial file to be read and displayed.

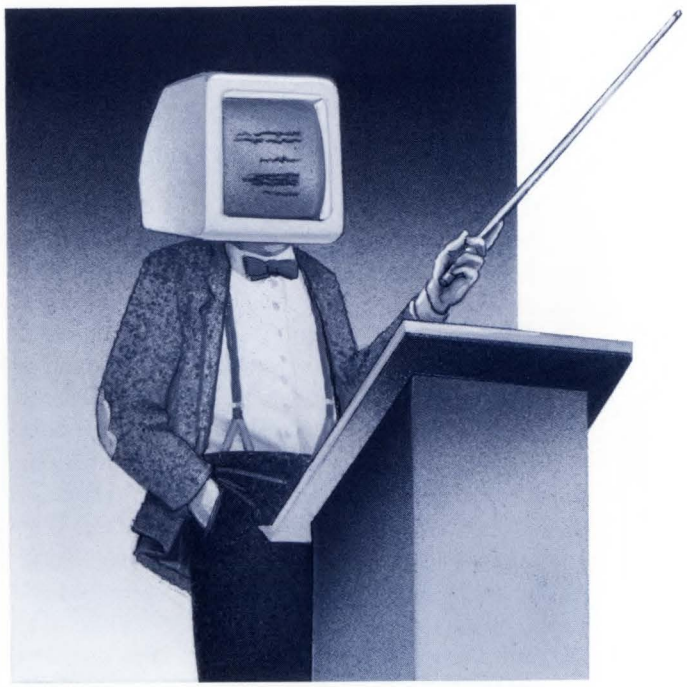
The command line options /p and /s control the drive and file name to be used for the PE2 profile and spill file, respectively. The name of the file follows immediately after the one-character command name. For example, the line

```
PE2 /pd:pe.pro /s\temp\petmp test
```

starts PE2 and displays the file named "test" from the current directory of the default drive. The profile will be taken from the file "pe.pro" in the current directory on the D drive. The spill file, if it is needed, will be named "petmp.xxx" and will reside in the directory \temp on whatever is the default drive at the time the spill file is created. Here as elsewhere, path names may use either forward or backward slash (/ or \) to delimit their components. The default values for /p and /s are /ppe2.pro and /spe2.tmp, respectively.

The command line option /q (quiet) tells PE2 not to display the initial logo screen. Whether or not the logo screen is displayed, a copyright notice will always appear in the message area of the screen when the editor is first started. It will disappear after the first keystroke.

The command line option /r reserves storage for second-level command interpreters loaded by the PE2 command "DOS". The format of the /r option is /rnnn, where nnn is a decimal integer. The number given as the /r option is the number of thousand-byte units PE2 will reserve for DOS commands. PE2 will never use the reserved area to store active files or macro definitions. The default (and minimum possible) amount of storage to be reserved is 2KB, written as /r2. PE2 will round the specified memory reserve size according to how much is actually available. Thus, /r999 will reserve as much as possible for DOS commands after the PE2 code and data segments have been allocated. The PE2 DOS command



(described below) returns all dynamic storage currently not in use by PE2 to the operating system before loading the second-level command interpreter, so it should not be necessary to use the /r option very often.

The command line option /b forces PE2 to perform all of its screen updates using BIOS. This may be useful when running in certain environments. Screen update performance is approximately 20 times slower when using the /b option.

Demonstration Mode

You can set PE2 to display or demonstrate each command of a key definition or macro as it executes. The [DEMO FAST] command turns off the deferred screen update (mentioned above) to allow the action of each keystroke within a macro to be displayed as it occurs. The [DEMO SLOW] command is similar, but inserts a short delay between each automatic keystroke so that the operation of a macro can be better observed. The [DEMO END] command turns off the demonstration mode. Using these primitive commands, fairly comprehensive demonstrations and tutorials can be constructed, especially when combined with the split screen feature.

You can also use the demonstration mode to debug keystroke macros. Suppose you have assigned a complicated key definition to key c-f1. The command "[DEMO SLOW] [key c-f1] [DEMO END]" will execute the actions of the key c-f1 slowly enough for you to understand how it works (or fails to work).

Improvements to Existing Commands
Several of the commands in PE1 have been enhanced with new options or more rational meanings in PE2.

MARK: The MARK command operations have been modified. The first MARK command of a given type (line, block, character) behaves just as in PE1. The second MARK of that type within the same file marks all of the text between the first and second mark points, just as in PE1. Issuing a third MARK command in PE2 does not result in an error as it did in PE1. Instead, PE2 assumes you mean to alter the bounds of the marked area. It marks the text between the first and third marks, and discards the location of the second mark. Each subsequent MARK renews the bounds of the marked area, with the first place marked always remaining as one of the boundaries. The third and subsequent marks can be either before or after the first and second marks.

Display Color 80: When used on a color display, PE2 uses different colors for each of the fields on the display. Use the "SET DISPLAY COLOR 80" command to enable PE2 for color.

EDIT, FILE, SAVE: The EDIT, FILE and SAVE commands now take an option TABS as an alternative to the PE1 option NOTABS. This option works in conjunction with the new SET BLANKCOMPRESS and SET TABEXPAND commands (see the section below) to determine the handling of tab characters.

CHANGE, LOCATE: A new option 'm' in the CHANGE and LOCATE commands restricts the area to be searched to the currently marked area, if any. This makes it possible to change all occurrences of a word within a single, marked paragraph without affecting words outside of the marked paragraph. The new option is indicated by ending the CHANGE or LOCATE command with 'm' (for Mark). For example,

```
c/martian/venetian/*m
```

changes all instances of the word "martian" that start after the current cursor position and are within the marked area to "venetian" without asking for confirmation after each instance is found.

SET SEARCHCASE xxxxx: The CHANGE and LOCATE commands can optionally respect case in performing searches. The command SET SEARCHCASE EXACT requires that the data in the file match the search string exactly. SET SEARCHCASE ANY (the default) sets these commands to behave as they do in PE1 — to ignore the case of strings in performing searches. The command ? SEARCHCASE displays the current setting of SET SEARCHCASE on the command line.

[UP], [DOWN], [LEFT], [RIGHT]: The [UP], [DOWN], [LEFT], and [RIGHT] commands accept an optional number as an argument in PE2. For example, you may now use the command "[RIGHT 3]" instead of "[RIGHT] [RIGHT] [RIGHT]". PE1 contained some special cases of directional movements in increments greater than one: [UP4], [DOWN4], [LEFT8], [RIGHT8], [LEFT40], and [RIGHT40]. These special cases are no longer necessary in PE2.

LINE and COLUMN: PE2 also contains the commands LINE nnn and COLUMN nnn, which position the cursor to the indicated line or column number. Line and column numbers that are outside the bounds of the file cause the cursor to be positioned at the appropriate file boundary.

[TABWORD] and [BACKTAB WORD]: To assist users in doing word processing with Personal Editor, PE2 implements the [TAB WORD] and [BACKTAB WORD] commands slightly different from PE1. In PE2, these commands span lines of the file, not stopping at the end of each line as in PE1. Thus, starting at the top of a file, successively iterating [TAB WORD] will move the cursor word by word toward the end of the file.

The algorithm for screen updates was substantially revised. PE2 defers all changes to the screen until all commands in a macro or a key definition have finished executing.

Status Line: If changes have been made to a file since the last SAVE, PE2 will highlight the display of the file name on the status line.

Insert Mode: In PE1, insert mode (as opposed to replace mode) sometimes inserted spurious blanks at the end of lines. For example, typing a single character on a new (empty) line while in insert mode caused PE1 to put two characters on the line: the one typed plus a blank. This was most annoying when typing locate commands: "l/xyz" would search for "xyz " instead of "xyz". PE2 corrects this deficiency.

New Commands

This section describes new commands that have been added to PE2.

SET BLANKCOMPRESS and SET TABEXPAND: By default, PE1 compresses runs of blanks to tab characters on output and expands tab characters to blanks on input. The syntax to override the default is difficult to use and impossible to place on a function key. Two new commands in PE2, SET BLANKCOMPRESS ON/OFF and SET TABEXPAND ON/OFF, control how PE2 handles both runs of blanks and tabs.

Setting both options to ON (the default) yields the same behavior as PE1 — blanks are compressed to tabs on output and tabs are expanded on input unless the NOTABS option is given on the EDIT, FILE, or SAVE command. If SET BLANKCOMPRESS is OFF in PE2, no strings of blanks are converted to tabs. Blanks are stored as blanks. You can override this using the new TABS option just like NOTABS is used in PE1 (e.g. FILE xxxxx TABS). Similarly, SET TABEXPAND OFF disables expansion of tab characters to runs of blanks when a file is read by the EDIT command, unless the TABS option appears in the EDIT command.

DOS: PE2 permits loading secondary command interpreters using the new command DOS. The DOS command compresses all active files, but continues to hold them in memory. Any memory not used for storing the compressed files is returned for use by the second-level command interpreter. The amount of storage can be determined while in PE2 by issuing the new command ? MEMORY. PE2 then searches the current environment string for the CMDPROC= parameter, and loads and executes the command interpreter named there (usually COMMAND.COM).

The second-level COMMAND.COM can be used to perform any DOS command. When the command EXIT is issued to the new command interpreter, PE2 will resume exactly where it left off. PE2 guards itself against modification by other programs by computing a checksum on its code and data every time control is passed to a secondary command interpreter. The checksum is verified when the command interpreter exits.

[PUSH MARK] and [POP MARK]: PE2 allows the bounds of the current marked area to be pushed onto a stack, then restored later. This permits keystroke macros that do not disturb existing marks. The [PUSH MARK] and [POP MARK] commands save and restore marked areas, with a maximum stack depth of six (five saved to the stack plus the current mark). The [UNMARK] command unmarks only the current mark, leaving intact the marks saved in the stack. A new command, [CLEAR MARKS], unmarks all marked areas. All operations on marked areas apply only to the current mark, the one that is displayed (the top of the stack). However, the boundaries of marked areas that have been pushed are adjusted in response to changes in the file just as they would be if the marked area were the current mark. For example, suppose lines 100 through 105 are marked in a file. If lines 1 through 10 are deleted, in order for the marked area to still contain the same data as before, its boundaries must be relocated to lines 90 through 95. PE2 does this relocation whether or not the marked area is on the top of its stack.

[BEGIN WORD] and [END WORD]: The [BEGIN WORD] and [END WORD] commands move the cursor to the first and last characters, respectively, in the word containing the cursor. They are useful in word processing applications. For example, deleting the word pointed to by the cursor can be done with the definition:

```
def a-w = [PUSH MARK] [END WORD]
          [MARK CHAR] [BEGIN WORD]
          [MARK CHAR] [DELETE MARK]
          [POP MARK]
```

[LEFT MARGIN], [RIGHT MARGIN] and [PARAGRAPH MARGIN]: PE2 has the new commands [LEFT MARGIN], [RIGHT MARGIN], and [PARAGRAPH MARGIN]. These move the cursor to the column on the current line corresponding to the left, right, and paragraph margins as set by the SET MARGINS command.

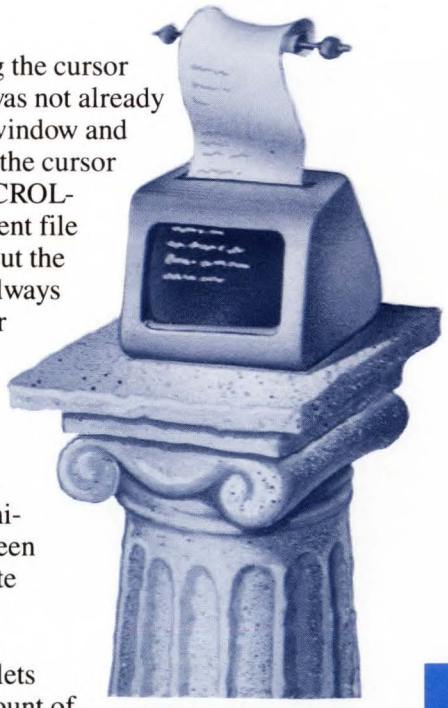
[CENTER IN MARGINS]: Text may be centered in PE2 using the [CENTER IN MARGINS] command. It operates on an area defined by line marks. PE2 centers each line in the marked area between the left and right margins established with the SET MARGINS command.

[COPY TO COMMAND] and [COPY FROM COMMAND]: The [COPY TO COMMAND] command copies a marked line in the data area to the command line. The inverse [COPY FROM COMMAND] command inserts a new line in the data area following the line containing the cursor, then fills it with a copy of the command line.

SET BACKUP: The command SET BACKUP n controls the amount of data to be kept in the .unnamed file. In PE2 (and PE1) all of the lines changed in response to a keystroke are saved as a group in the internal file .unnamed. Block move commands can be used to recover the original contents of lines deleted or changed inadvertently. As each group of new lines is added to .unnamed, the oldest group is deleted. The SET BACKUP command controls the number of groups of lines saved in .unnamed. For n=5, the default, PE2 will behave exactly like PE1 with respect to .unnamed. For smaller values of n, fewer groups of lines will be saved (and hence less memory will be consumed by this function). SET BACKUP 0 disables the saving of lines in .unnamed. PE2 supports as many as ten groups of backed-up lines.

SET HSCROLL: In PE, when the cursor attempts to leave the screen, the data on the screen is shifted. When the cursor moves vertically off the screen, the shift amount is one row. In PE1, when the cursor moves horizontally off the screen, the shift amount is half the width of the screen. PE2 contains a facility to allow the horizontal shift amount to be either half the screen width or a single column. SET HSCROLL OFF is the default, and gives the same horizontal scroll behavior as PE1. SET HSCROLL ON sets the horizontal scroll increment to a single column. The command ? HSCROLL displays the current setting of SET HSCROLL.

SCROLLxxxxx: PE2 has the four new commands SCROLLUP, SCROLLDOWN, SCROLLLEFT, and SCROLLRIGHT. The syntax of these commands is identical to the UP, DOWN, LEFT, and RIGHT commands (optional number argument; default of one). They move the cursor through the file exactly like UP, DOWN, LEFT, and RIGHT, but change the display differently. UP 1 moves the cursor up one line in the file, moving the cursor up on the screen if it was not already on the top line of the window and scrolling the screen if the cursor was on the top line. SCROLLUP 1 moves the current file position up one line, but the data on the screen is always scrolled and the cursor remains in the same place on the screen (unless the top of file line is visible). The other SCROLLxxxxx commands behave similarly, scrolling the screen unless their appropriate edge is in sight.



? DISKSPACE: PE2 lets you determine the amount of space remaining on a disk drive with the ? DISKSPACE command. It displays the number of bytes remaining on a drive. This command accepts an optional drive letter, without the colon suffix, as in "? DISKSPACE C".

[PRINT MARK]: The [PRINT MARK] command prints the contents of the current marked area. The existing [PRINT] command prints an entire file. Either command can be terminated early by typing Ctrl-Break.

[BEEP]: The [BEEP] command sounds a short tone on the speaker.

[CLEAR MESSAGE]: The [CLEAR MESSAGE] command clears the error message area. It is useful to suppress the messages that may normally occur during macro processing.

***:** Comment lines may be included in a command file by placing an asterisk (*) in the first column of a line in the command file. PE2 will not process the contents of the line.

An Overview of IBM BASIC Compiler 2.00 (Part 2)

John Warnock
IBM Corporation

Editor's note: This is the second in a two-part series introducing IBM BASIC Compiler 2.00. Part 1 described changes in BASIC Compiler 2.00 and differences between the BASIC Compiler and the BASIC interpreter. This part discusses modular programming techniques, index files, and the Library Manager. This article is adapted from the IBM Personal Computer Seminar Proceedings.

Index Files

IBM Personal Computer ISAM Files
Normally, BASIC supports two types of data files: *sequential* and *random access.* These two types are sufficient for most applications. The major limitation of these two file types, however, is that they do not let you access the records in the file by their content. You must either search the entire file until you find the record you want, or you must know the number of the record you want.

The Indexed Sequential Access Method (ISAM) is a library of subroutines that lets you access files sequentially or randomly by an *index.* ISAM files supported by BASIC Compiler 2.00 have a special format, consisting of a data file *plus* an index. An index serves as the table of contents to a file. The index contains the information that you use to request a record, plus the relative record number that ISAM uses to find the record.

With ISAM, you can access data based on the *content* of your records. For example, if you want to read the record that contains information about product number 34056-J, delete any records with information on employee T.R. James, or update the record containing information about the price and availability of marble, ISAM provides a fast way to do so.

ISAM Glossary

ISAM Terms and Concepts
Here are a few basic ISAM terms that you will encounter:

File Handles
The number used by ISAM to refer to a specific ISAM file. Although physically an ISAM file is two files (a data file and a key file), there is only one file handle for each data/key file pair. ISAM returns the file handle each time you open a file.

Data Record
The basic unit of data in an ISAM file as in BASIC random files. Most ISAM subroutines operate on one record at a time. Generally, there will be a record in your ISAM file for each employee, part or whatever piece of information you are storing. An ISAM file can contain any number of records.

Data Type
The Type of value that is stored in a particular variable. BASIC ISAM supports five data types: integer, string, numeric, single-precision and double-precision.

Field
Part of the data record containing a single value of a particular data type. For example, an ISAM file may contain records that are composed of fields for name, address and phone number. A data record can consist of any number of fields.

Key Field
The field that contains the value ISAM uses to determine where the record goes in the file.

Key Value
The value stored in the key field. For example, the key field might have 20 bytes reserved for employee name and the name stored there may be John Doe. John Doe is the key value for that key field.

Key Handles
The number used by ISAM to refer to a specific key in an ISAM file. The key handle is assigned by the programmer in the field description when the ISAM file is created. Key handle values range from 1 to n, where n is the number of keys. There does not have to be any physical relationship between the key handle values and the record layout, but it is a good idea to assign key values from the lowest to highest part of the record.

ISAM File Structure

Physically, each ISAM file is two files: a data file and a key file. File names cannot exceed 8 characters. Conventionally, data file names end with a .DAT extension and key file names end with a .KEY extension. Both files must be present to use ISAM.

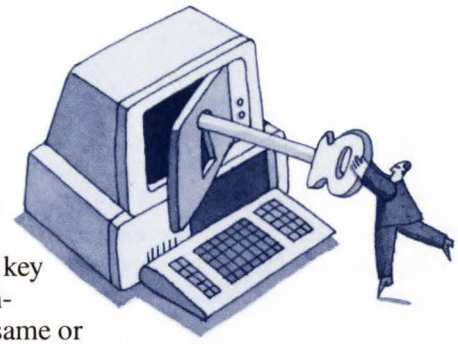
The data file usually consists of data records and a data dictionary. The data dictionary, which is at the beginning of the data file, contains binary descriptions of records in the file. Whenever you create an ISAM file, you give ISAM information about how your data is formatted, such as where data fields start and end, what type of data is contained in a field, and if it is acceptable to have duplicate key values in the file. This information is stored in the data dictionary.

The Indexed Sequential Access Method (ISAM) is a library of subroutines that lets you access files sequentially or randomly by an index.

The key file contains the indexing information (where the fields are, what type of data they contain) that ISAM uses to access the data in the data file. A key field is a data field that has been identified and described to ISAM as the index. Using ISAM, you can access records in the data file according to the value contained in the key field. ISAM gets this information from the data dictionary in the data file. The indexing is in the form of a B-tree, a special kind of index that points to the records in the data file. There is one B-tree in the key file for each key that you specify for that data file.

Whenever you access an ISAM file, ISAM automatically obtains the information it needs from the data and key files and writes the appropriate information to each file. For example, when you write a record to an ISAM file, ISAM writes the data record to the data file and the key value and relative record information to the key file.

When choosing the key for a file, you should make the value of that key unique for each record in the file. This may help avoid any confusion when searching the records in the file for a particular key. For example, social security number is a good field to use as a key because everyone has a unique social security number. Using last name as a key field can cause problems if you have two people named Smith.



Split Keys

There is a special type of key, called a split key, that contains more than one field. Components of a split key can be adjacent or non-adjacent fields of the same or different data types, and may or may not be keys themselves.

If a component field of a split key is also a key, that field's description must be given twice: once to describe it as a key field and once to group it with the other components of the split key. This type of field also has more than one key handle: one handle of its own and another handle that is the same as the other components of the split key.

When key values are compared (to determine the order of records or to determine if values are equal), the split key components are compared according to the order in which they were declared in the key description. If the two components have equal values, the next component in the split key is compared. This is repeated until a difference is found.

Note: Split keys cannot be used with segmented records.

ISAM Record Types

There are two types of ISAM data records: non-segmented and segmented. These two record types cannot be mixed in one file.

Non-segmented Records

Non-segmented records are the type most often used. They contain key fields and data fields that have fixed sizes. However, they may contain one field that is variable in length, as long as the variable length field is the last field in the record.

Segmented Records

ISAM data files contain either segmented or non-segmented records. These record types cannot be mixed in one file.

Segmented records, the second type, support key fields that can vary in size. Segmented records are usually used to contain variable-length strings. You can use variable-length strings without using segmented records by setting the length of your string field long enough to hold the longest string you are using.

Note: You should use segmented records only if it is very important to minimize the amount of storage space used for variable-length fields.

The address of a key field is given by a segment number and an offset. For non-segmented records, the segment number is 1. In segmented records, the segment number acts as an index to a segment table, which must be inserted in front of each record. The segment table is an array of 16-bit offsets; this offset is the number of bytes from the start of the record to the start of the segment.

For a given key n, the address of the key is the address in the nth entry in the segment table, plus any offset within the segment itself. A field length of zero indicates that the field length equals the length of the entire segment.

You should describe each field in the record when you create an ISAM file. This provides an easy way to identify each file and its contents.

The number of segments, the segment table and offsets within segments must be supplied in the record and field descriptions when the file is created. The segment table must be maintained by the application programmer. For this reason, you should use segmented records only if variable-length key fields are needed. Often, all fixed-size record fields are placed in the first segment, and each variable-length string field is placed in its own segment.

The following diagram illustrates a three-segment record with three keys:

- Key 1 is fixed-length, begins at offset 10 into the segment and has a length of 4.
- Key 2 is in segment 2, begins at offset 0 and occupies the entire segment.
- Key 3 is in segment 3, begins at offset 0 and occupies the entire length of the segment.

If a data file contains segmented records, it is not necessary for each record to contain the same number of segments. All segments that contain keys, however, must be present in each record. If a segment that contains a key is missing from a record, the status code, IXSTAT = 10 (key not found), is returned.

Note: Segmented records cannot contain split keys.

Writing an ISAM Application

The ISAM interface is designed to make access to ISAM files as simple as possible. In general, ISAM file access is similar to BASIC random I/O. Specific ISAM subroutines are called to open and close ISAM files, to find records within a file and to read, write, delete or rewrite data.

Note: MAIL.BAS is a demonstration program included on the ISAM diskette as an aid in developing your own ISAM programs.

There are six basic steps in creating and using ISAM files:

1. Install ISAM.EXE.
2. Open an ISAM file.
3. Seek to (search for) some record in the file.
4. Operate on the data.
5. Check the results.
6. Close the ISAM file.

Using the FIELD Statement

ISAM files are similar, in some ways, to BASIC random files. When using ISAM files, you must establish a data buffer. You can do this by using the FIELD or COMMON statements.

The following example establishes a buffer for an ISAM file with a 10 character field for name and 25 character field for address:

```
OPEN "NUL" AS #9 LEN=35
FIELD #9, 10 AS NAME$, 25 AS ADDR$
```

The OPEN statement is necessary because a file must be open before you can allocate the data buffer with the FIELD statement. The file is opened as NUL because the buffer is written to the ISAM file, not the BASIC file. This means you need to locate the data buffer for ISAM.

To get the pointer to this data buffer, use the following procedure:

1. Insert the following function definition into your program.

   ```
   DEF FNSADD!(VPTR)
   FNSADD!=PEEK(VPTR+3)*256.0 +
      PEEK(VPTR+2)
   END DEF
   ```

   This function accepts the pointer to a string descriptor as input and returns the address of the string.
2. Execute the following:

   ```
   PDATREC!=FNSADD!(VARPTR(NAME$))
   ```

This places the address of the data record into PDATREC!.

Using the COMMON Statement
If your data record contains only numerical values, you can establish the data buffer using a COMMON statement. For example, if the data record consists of IDNUM and PHONENUM!, the following statement establishes the data buffer:

```
COMMON /DATA.REC/
  IDNUM,PHONENUM!
```

Once this is done, you can get a pointer to the data record with the following:

```
PDATREC! = VARPTR(IDNUM)
```

Note: Remember, you can use this form *only* if your data record consists solely of numerical values.

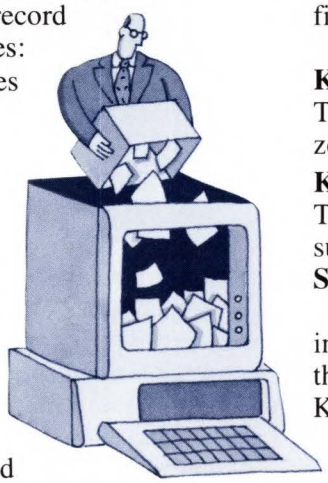
Record Description
The record description tells how many keys are in the record, whether the record is segmented or non-segmented, and the minimum record allocation. This information is given to ISAM as the array **Rdes.**

Rdes(1) = number of fields
The number of fields is declared in the field descriptor array. This number includes both key and non-key fields and components of split keys.

Rdes(2) = segment-flag
If the record is non-segmented, segment-flag = 0. If the record is segmented, segment-flag = 1.

Rdes(3) = minimum record allocation
If Rdes(3) = 0, the minimum record allocation defaults to eight bytes: five bytes of data and three bytes of overhead.

When a record is rewritten over a record that is too small to contain the new record, ISAM makes the old record into an indirection record. The indirection record points to the location of the new, larger record. By using indirection records, ISAM avoids having to change every key that pointed to the old record location. To make sure that every record is big enough to hold an indirection record, the minimum record allocation defaults to eight bytes.



Field Description
Whenever you create an ISAM file, you must describe each key field that you are using; this is the information used to build key files. You also can describe non-key fields. ISAM puts this information in the data dictionary at the beginning of the data file. Whenever a file is opened, its data dictionary is loaded into memory from the data file.

If you are using files created by IBM Personal Computer Sort version 1.00, you should be aware that some of these files do not have a data dictionary. When using these files, you must specify the field description each time you open the file.

You should describe each field in the record when you create an ISAM file. This provides an easy way to identify each file and its contents. Complete field descriptions can also be used by other utilities to access field information.

Creating A Key Descriptor
Field descriptions are given to ISAM as a nine-integer array, **Kdes.** The parameters that you must supply for each field you describe are explained below.

Kdes(1) = pointer-to-field-name
This is a pointer to a buffer that contains the name of the field. The field name must be less than or equal to 40 characters. If no field name is supplied, this pointer must be null. Field names can be used by utilities, such as general file dump utilities, to access fields in a data file.

Kdes(2) = 0
This is a reserved word area. It must be initialized to zero.

Kdes(3) = data-type
This is the data type of the field, which is set by supplying one of the following words: **INTEGER, STRING, NUMERIC, SINGLE, DOUBLE.**

You must use the $INCLUDE metacommand to include the file ISAM.INC in your program to set these values. Once this file is included, you can set Kdes(3) as in the following example:

```
KDES(3) = INTEGER
```

Kdes(4) = segment-number
The number of the segment containing the field. For non-segmented records, this number is always 1.

Segments are numbered from 1 to n where n is the number of segments. Segment 1 is the first segment in the record and segment n is the last. Each segment can contain many fields but no field can span more than one segment.

Kdes(5) = field-position
This is the position from the beginning of the segment to the beginning of the key field (the offset of the field in the segment). The first byte in the segment is numbered 1.

Together, Kdes(4) and Kdes(5) comprise the field address.

Kdes(6) = field-length
This is the length of the field in bytes. A zero length field indicates that the field size is from the field segment position to the end of the segment. If the field length is variable, this number should always be zero.

Kdes(7) = key-handle
Any value between 1 and n where n is the number of keys. The convention is to assign key handles beginning with 1 and starting with the leftmost byte in the record. Using this convention makes it easier to remember key handles. Key handles can be determined at run time by using the IGETKD procedure to fetch key field descriptions.

If you are defining a non-key field, this number is 0.

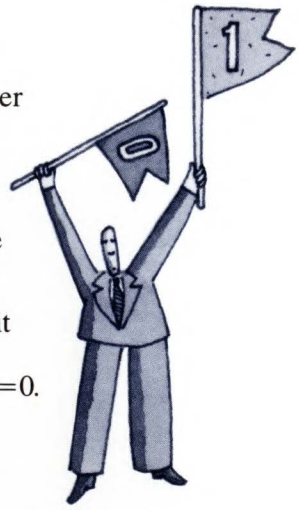
Kdes(8) [high byte] = duplicates-allowed flag, descending flag, and case-insensitive flag

- The duplicates-allowed flag indicates whether duplicate values are allowed for this particular field.
- The descending-flag inverts the meaning of comparisons performed on this field. The result is that the records are inserted into the key set in descending instead of ascending order. It is most useful with split keys where the ordering of the different components might need to be inverted.
- The case-insensitive-flag causes string-based data types to ignore differences in case (for example, the values "FiRst" and "first" would be equal).

The duplicates-allowed flag = 1, the descending flag = 2, and the case-insensitive flag = 4.

Add the values of the desired flags together and enter that number as the high byte. For example, to set the duplicates-allowed and case-insensitive flags, use:

```
Kdes(8) = (256 * (1 + 4)) + field-mode
```



Kdes(8) [low byte] = field-mode
Tells if the field is a key, and if so, whether it is a split key.

If the field is a non-key field, field-mode=0. If the field is a non-split key field, field-mode=1. If the field is a component of a split key, field-mode=2.

Kdes(9) = filler
This is a reserved word area. It must be initialized to zero. The following example shows typical record and key descriptors:

```
DIM RDES(3), KDES(9)
$INCLUDE:'ISAM.INC'
RDES(1) = 1                 '1 Key field
RDES(2) = 0                 'Non-segmented
RDES(3) = 8                 'Minimum record length
FIELDNAME$="ID
  NUMBER"
KDES(1) = VARPTR
  (FIELDNAME$)              'Field name pointer
KDES(2) = 0                 'Reserved—Set to zero
KDES(3) = INTEGER           'Data type—Integer
KDES(4) = 1                 'Segment 1
KDES(5) = 1                 'Position 1
KDES(6) = 2                 'Length of field in bytes
KDES(7) = 1                 'Key number is 1
KDES(8) = (1*256) +1        'Duplicates allowed—
                             Nonsplit key
KDES(9) = 0                 'Reserved—Set to zero
```

Modular Programming Techniques

Structuring Modular Programs
Because BASIC Compiler version 2.00 allows you to compile and link separate subprograms, you have a flexible environment for structuring large programs. The subprogram, module and named COMMON block structures let you construct libraries of compiled BASIC modules. Each module within a library can consist of one or more subprograms. Parameters passed with the CALL statement and named COMMON blocks provide communication among modules.

Modular programming has three major benefits:

1. Ease of understanding—each subprogram or module has a specific task that it performs on a small number of parameters or common variables. This makes the overall program much easier to understand.
2. Subprogram independence—Because the communication paths among subprograms are isolated and well-defined, a given subprogram is less dependent on other subprograms. This allows the subprograms to be developed independently, even by different programmers. It also permits subprograms to be reused across several applications.
3. Program flexibility—If the dependencies among subprograms are minimized, a given subprogram may be completely rewritten or replaced by one with a better algorithm, as long as its overall results do not change. Such modularity enhances the overall flexibility of the application program by allowing it to be tailored for another system or user. An example is recompiling a single module to change a report heading and relinking the program.

Because BASIC Compiler version 2.00 allows you to compile and link separate subprograms, you have a flexible environment for structuring large programs.

Named COMMON Blocks

BASIC Compiler version 2.00 supports named COMMON blocks as one way of supporting larger programs. COMMON blocks create data segments that can be accessed by BASIC program segments. Named COMMON blocks are declared by using the blockname option with the COMMON statement.

Named COMMON blocks differ from unnamed (blank) COMMON statements (where blockname is not specified) in one key aspect. Items (variables and arrays) in named COMMON blocks are not preserved when you chain to another program with the CHAIN statement. To protect the information in a named COMMON block you must use the same blockname in the chained program and use the same variable types and sizes in the same order. The variable names need not be the same, however. For example, chaining between **A,B,C(2)** and **E,F,G(2)** would be valid, but **A,B,C(2)** and **E,F(2),G** would not.

There are two other ways of sharing items declared in COMMON blocks. One is for the independent modules to communicate through the CALL statement. This lets named COMMON blocks be used for inter-module communication without chaining. The other way to share items between programs is to omit the blockname. Items that are listed in the regular COMMON statement (no blockname specified) can still be accessed by another chained program as in interpreted BASIC.

The Library Manager

The IBM Library Manager allows you to construct and edit object module libraries. Object files and other library files can be added to a library and object modules can be removed and erased from a library.

Command Line Format

The format of the command line is:

```
LIB [library-file +
    [pagesize] + operations +
    [,[list-file]][, +
    [newlib]][;]]
```

where

library-file is the name of a library file.

page size is an optional switch of the form,

"/pagesize::N" or "/p::N"

where *N* equals:

16, 32, 64, 128, 256 or 512.

By default, libraries under IBM DOS are always multiples of 512 byte blocks. Object modules always start at the beginning of a new block. A block is also called a page. If the size of the object module is less than a block, the rest of the block is filled with null bytes.

When you specify value for page size in the command line, the library being created or modified contains *N* byte pages.

The size of the library that you are creating or modifying can increase when you specify larger values for page size. However, the time it takes to link the library decreases when you use larger page size values.

The default value for the page size switch is 512 if the library file is being created, or the current page size if the library file is being modified.

Note: Version 2.30 of the Linker is included with this version of the BASIC Compiler 2.00. Previous versions of the Linker cannot recognize page size values less than 512. Therefore, you should always use the latest version of the Linker.

operations is a list of operations to perform. This list contains an operator plus the name of the file you are adding. The default is an empty list; no changes occur. See "Operators," later in this section for a description of the operators.

list-file is a filename where a cross reference listing will be placed. No default extensions are used.

The default for [,*list-file*] is no list file; a cross reference is not generated. You are asked for this entry if it is left empty.

newlib defines the name of a library file to be created with the changes specified by the *operations*. The default is the same name as the library file. If you use the default, the original file is renamed to have the extension "BAK" instead of "LIB."

The command line can be broken by a carriage return at any point. You are asked for the remaining parts of the command line. If a semicolon ends any field after the library file name, the remaining fields take on its default value. If you just specify LIB, you are asked for all entries.

Note: You can have a device identification before any of the entries that you specify in the command line.

Operators

The operators recognized by the Library Manager are:

+ Adds the contents of an object file or a library file.

– Erases an object module.

* Retrieves an object module and copies it into a file whose name is the specified module name plus the extension .OBJ.

These individual operators can be combined to perform more complex operations. For example:

± Replaces an object module with the contents of the object file of the same name (plus ".OBJ").

–* Removes an object module and at the same time erases it.

Many operations may be performed at once. If you want to specify operations on more than one line, follow your last operation with an '&' and a carriage return.

First, erasures and removals are performed in the order in which the specified object modules occur in the library. Then, additions are performed in the order you specify.

To add the file TEST.OBJ to the library BASIC.LIB without producing a cross reference, type:

```
LIB BASIC.LIB+TEST.OBJ;
```

Note that the following is the same as the preceding example:

```
LIB BASIC+TEST;
```

Extensions are optional, and they default to .OBJ if omitted. If you are using a library file that is in the operations list, you must specify the .LIB extension.

To erase TEST from BASIC.LIB, type:

```
LIB BASIC-TEST;
```

To replace TEST in the library with a newer version, type:

```
LIB BASIC-+TEST;
```

If you want to make the same changes but put the changes in a new library called BASNEW.LIB, type:

```
LIB BASIC-+TEST,,BASNEW
```

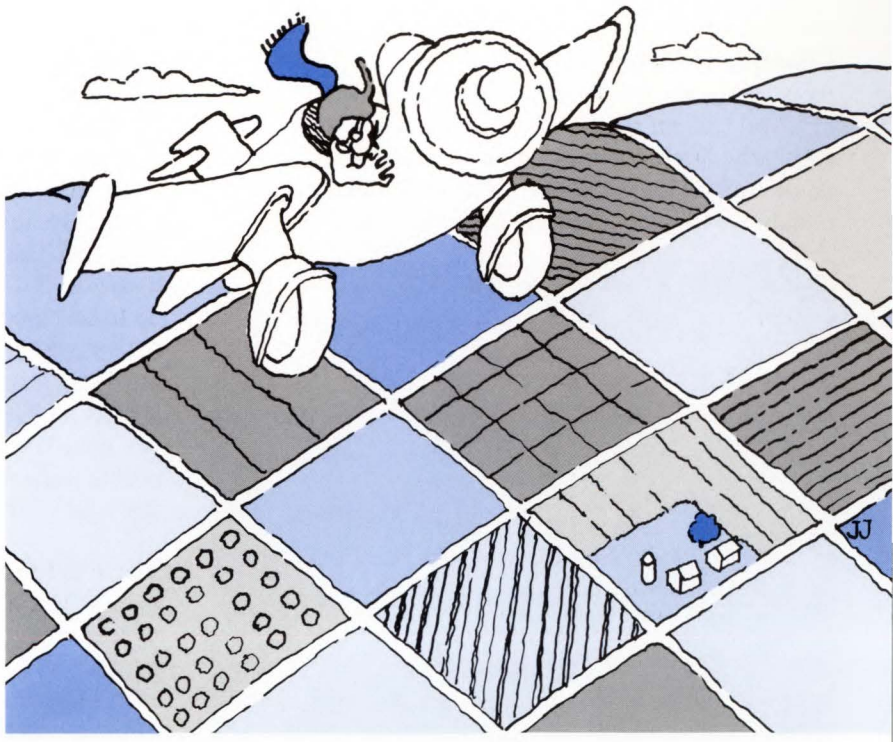
If you want to create a library of object modules, type:

```
LIB MYSUBS+FILE1.OBJ+FILE2.OBJ+
    ...+FILEN.OBJ.
```

You are asked for the listing file.

Multiple Field Definitions In BASIC

Martin Bryskier
IBM Corporation



Data files usually consist of many records, and records usually contain many fields. A field is simply a related group of adjacent characters (bytes) of data. For example, in the vast Social Security data base, your individual record might contain (among other things) your Social Security number, name, address, birth date, earnings each year, and amount contributed toward Social Security each year. Fields vary in length — the field containing your name occupies more space than the field containing your birth date.

Computer programming languages have ways of defining fields within records. This article concentrates on how to define fields in IBM Personal Computer BASIC, and emphasizes defining many fields for one record by using several statements.

The OPEN Statement

Suppose that on a diskette in drive B you have a data file named SOCSEC.DAT. (Here the extension .DAT is used to make it obvious that the file contains data. However, the extension can be almost anything you wish.) In a BASIC program you would gain access to this file by using an OPEN statement such as the following:

```
10 OPEN "R,"1,"B:SOCSEC.DAT",346
```

In this OPEN statement:

"R"	Indicates a random access file, which means you have direct access to individual records.
1	Designates this file as file number 1 within this BASIC program. All further statements that access this file will use the number 1 to point to this file.
"B:SOCSEC.DAT"	Is the pointer to the drive and to the file itself. File number 1 is equated to SOCSEC.DAT through the OPEN statement.
346	Is the length in bytes of an individual record in this file. It is also the length of the input/output buffer for this file. BASIC can work with records up to 32,767 bytes.

The OPEN statement actually initializes the input/output buffer. When you issue a GET command to read a record from the file, that record goes into the buffer. Similarly, when you place a record in the buffer and issue a PUT command, that record is written to the file.

Specify Buffer Size When Calling BASIC

The OPEN statement above specifies a record length of 346 bytes. BASIC assumes that records from random files are 128 bytes long, and BASIC sets up its work areas accordingly. When you have a record longer than 128 bytes, you have to notify BASIC at the time you invoke BASIC.

To do this you must invoke BASIC using the option /S:bsize (see your *BASIC Reference* manual for more information). This parameter sets the buffer size that BASIC uses for random files. In the case above, bsize is 346, so you should invoke BASIC as follows.

```
BASIC /S:346 (plus other options you may
      require)
```

Example of Fields in a Record

Now that you have defined a 346-byte buffer in memory, you can define fields within those 346 bytes. Using the Social Security data base example (which is, of course, hypothetical), if an individual's record contains 346 bytes, it might look like this:

Bytes	Length	Contents
1-9	9	Social Security number (dashes omitted)
10- 29	20	Last name
30- 49	20	First name
50- 69	20	Middle name
70- 75	6	Birth date (YYMMDD)
76-110	35	Current street address
111-135	25	Current city
136-137	2	Current state (postal abbreviation)
138-146	9	Current zip code (9 digits)
147-155	9	Social Security number of spouse
156-159	4	Year that Social Security contributions began
160-168	9	Total earnings from beginning year through 1973 (ddddddcc where d=dollars, c=cents)
169-176	8	Total Social Security contributions from beginning year through 1973 (ddddddcc)
177-185	9	Total earnings for 1974
186-193	8	Total Social Security contributions for 1974... (and so on)
313-321	9	Total earnings for 1982
322-329	8	Total Social Security contributions for 1982
330-338	9	Total earnings for 1983
339-346	8	Total Social Security contributions for 1983

There are 33 fields defined in this record. For the sake of simplicity, assume that every field is a string of characters (i.e., there are no fields set up for numeric data).

In order to work within a field, the field needs a name. Because your fields are all character strings, their names are string variables. String variable names end with the $ character. You can use names of up to 40 characters (including the $). You might want to use long names to be descriptive—for example, TOTAL_EARNINGS_THROUGH_1973$ or YEAR_CONTRIBUTIONS_BEGAN$. (Although you could certainly use much shorter names, this article purposely uses long names to illustrate a point that will be made soon.)

The FIELD Statement

In your BASIC program you use a FIELD statement to define fields. A FIELD statement contains the file number and a list of fields separated by commas. For each field the FIELD statement includes the number of bytes and the assigned field name. (A field name is also called a field variable. And, if, as in this example, the field variable is assigned to a character string, it is also called a string variable.)

For your Social Security example, your FIELD statement looks like this:

```
20 FIELD 1, 9 AS SOCIAL_SECURITY
   NUMBER$, 20 AS LAST_NAME$, 20 AS
   MIDDLE_NAME$, 20 AS FIRST_
   NAME$, 6 AS BIRTH_DATE$, 35 AS
   STREET$, ...
```

In the above FIELD statement:

- 1 is the file number, which refers back to the OPEN statement for the drive and name of the actual file.
- 9 AS SOCIAL_SECURITY_NUMBER$ defines a field of 9 bytes with the string variable SOCIAL_SECURITY_NUMBER$. Because this is the first field defined, the 9 bytes are assigned to bytes 1 through 9 of the buffer.
- 20 AS LAST_NAME$ defines the next field to be 20 bytes with the string variable LAST_NAME$. This field is assigned to bytes 10 through 29 of the buffer.
- and so on.

Entering Definitions for Many Fields

Theoretically you could continue adding fields to your FIELD statement until you have defined all 33 fields. However, a BASIC statement is restricted to a maximum of 255 characters. With your long descriptive field names you will reach 255 characters before you are able to define all 33 fields. (Long field names were used to make this point.)

You might think you can get around this restriction by splitting your field definitions into more than one FIELD statement. However, according to the BASIC manual, each new FIELD statement redefines the buffer from the first character position. Therefore, if your second FIELD statement were to look like:

```
30 FIELD 1, 4 AS YEAR_CONTRIBUTIONS
   _BEGAN$, 9 AS TOTAL_EARNINGS_
   THROUGH_1973$, ...
```

Bytes 1 through 4 would be assigned the string variable YEAR_CONTRIBUTIONS_BEGAN$, bytes 5 through 13 the string variable TOTAL_EARNINGS_THROUGH_1973$, and so on. Obviously, this is not what you want. How, then, can you define all 33 fields in their proper positions?

There is a way to define your string variables that will accomplish what you want to do. Look at this example:

```
20 FIELD 1, 9 AS SOCIAL_SECURITY_
   NUMBER$, 20 AS LAST_NAME$, 20 AS
   MIDDLE_NAME$, 20 AS FIRST_
   NAME$, 6 AS BIRTH_DATE$, 35 AS
   STREET$, 25 AS CITY$, 2 AS STATE$, 9
   AS ZIP_CODE$, 9 AS SPOUSE_
   SOCIAL_SECURITY_NUMBER$

30 FIELD 1, 155 AS FILL1$, 4 AS YEAR_
   CONTRIBUTIONS_BEGAN$, 9 AS
   TOTAL_EARNINGS_THROUGH_
   1973$, 8 AS TOTAL_CONTRIBUTIONS
   _THROUGH_1973$, 9 AS TOTAL_
   EARNINGS_IN_1974$, 8 AS TOTAL_
   CONTRIBUTIONS_IN_1974$

40 FIELD 1, 193 AS FILL2$, 9 AS TOTAL_
   EARNINGS_IN_1975$, 8 AS TOTAL_
   CONTRIBUTIONS_IN_1975$, 9 AS
   TOTAL_EARNINGS_IN_1976$, 8 AS
   TOTAL_CONTRIBUTIONS_IN_1976$,
   9 AS TOTAL_EARNINGS_IN_1977$, 8
   AS TOTAL_CONTRIBUTIONS_IN_
   1977$

50 FIELD 1, 244 AS FILL3$, 9 AS TOTAL_
   EARNINGS_IN_1978$, 8 AS TOTAL_
   CONTRIBUTIONS_IN_1978$, 9 AS
   TOTAL_EARNINGS_IN_1979$, 8 AS
   TOTAL_CONTRIBUTIONS_IN_1979$,
   9 AS TOTAL_EARNINGS_IN_1980$, 8
   AS TOTAL_CONTRIBUTIONS_IN_
   1980$

60 FIELD 1, 255 AS FILL4$, 40 AS FILL5$,
   9 AS TOTAL_EARNINGS_IN_1981$, 8
   AS TOTAL_CONTRIBUTIONS_IN_
   1981$, 9 AS TOTAL_EARNINGS_IN_
   1982$, 8 AS TOTAL_CONTRIBUTIONS
   _IN_1982$, 9 AS TOTAL_EARNINGS_
   IN_1983$, 8 AS TOTAL_
   CONTRIBUTIONS_IN_1983$
```

Here statement 20, which took 206 characters to enter, defines ten fields that occupy bytes 1 through 155. Statement 30, therefore, begins with a string variable FILL1$ that is defined to be 155 bytes long. (The name FILL1$ is arbitrary — you can select any name you wish.) FILL1$ is another definition for the first 155 bytes, but you will probably never refer to the field FILL1$. Its sole purpose is to mark off filler space so that the next string variable in statement 30, YEAR_CONTRIBUTIONS_BEGAN$, is assigned to bytes 156 through 159. After that, the remaining string variables defined in statement 30 will be assigned their correct positions.

The string variables in statements 20 and 30 add up to 193 bytes. Therefore, statement 40 begins with the string variable FILL2$, defined to be 193 bytes long. FILL2$ marks off 193 bytes so that the next string variable in statement 40, TOTAL_EARNINGS_IN_1975$, is assigned to bytes 194 through 202.

There is another restriction to be aware of: a string variable can be a maximum of 255 bytes long. When it is necessary to mark off more than 255 bytes of filler space in a single BASIC statement, you will have to use two string variables to mark off the space. Statement 60 illustrates this.

In this manner, using BASIC statements of maximum length 255 characters and filler variables of maximum 255 bytes, you can define whatever number of fields you require for your record, and you can span any record length up to 32,767 bytes.

Putting Data into Field Variables

After you read a record from a random file into the buffer, you can transfer data out of fields in the buffer and into string variables that are outside the buffer. In the above example, suppose you have separately defined a string variable BEGIN_YR$ that contains four bytes. You can then transfer into BEGIN_YR$ the contents of the field variable YEAR_CONTRIBUTIONS_BEGAN$ by using a simple LET statement (where the word LET is optional):

```
[LET] BEGIN_YR$ = YEAR_
      CONTRIBUTIONS_BEGAN$
```

However, you can't turn this statement around because you can't use a LET statement to transfer the contents of a separate string variable into a field variable. Instead you have to use the LSET and RSET statements (see the BASIC manual). These statements move data into a random file buffer in preparation for a PUT (file) statement. If the field variable is longer than the separate string variable, LSET left-justifies the string in the field, and RSET right-justifies it.

To do the opposite of the LET statement above, use

```
LSET YEAR_CONTRIBUTIONS_
     BEGAN$ = BEGIN_YR$
```

DOS Device Drivers

(Part 2)

John Warnock
IBM Corporation

Editor's note: This is the second part of an article about writing device drivers for DOS. This part examines the various functions that device drivers must support, and how to install and use device drivers. Part 1, published in the September 1985 issue of Exchange, *gave an overview of what device drivers do and how they do it.*

Device Driver Function Headers

Following are descriptions of several different function headers. The first five fields are always the same. The remaining fields vary by function.

INIT (Function 0)

The INIT function:

- Defines the number of units for DOS.
- Sets up each device.
- Sets the ending address of the driver.
- Sets the address for the pointer to the BIOS parameter block.
- Sets the status word.

The initialization function sets up the device driver within DOS. It is possible that the driver cannot set up the device. You should set up your initialization routine to allow for this. If an INIT routine determines that it cannot set up the device and wants to abort without using memory, it should do the following:

- Set the number of units to zero.
- Set the ending address offset to zero.
- Set the ending address segment to the code segment (CS).

Whether or not your INIT routine ends normally, it is advantageous to set the ending address offset to zero to ensure it lies on a paragraph boundary.

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 00H
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS
+13	1 byte	Number of units (block devices only)
+14	2 bytes	Ending address (offset) of resident code
+16	2 bytes	Ending address (segment) of resident code
+18	2 bytes	Indirect pointer to BIOS parameter block array (offset) (block devices only)
+20	2 bytes	Indirect pointer to BIOS parameter block array (segment) (block devices only)
+22	1 byte	Drive number (DOS 3.00 and DOS 3.10)

Figure 1. INIT (Function 0)

BASIC 3.10 Reference Chart

Compiled by John Warnock, IBM Corporation

This chart of BASIC 3.10 commands can be removed and used for future reference.

BASIC 3.10 Reference Chart

BASIC 3.10 Commands

BASIC	TYPE	DESCRIPTION
ABS(x)	Function	Returns absolute value of expression x
AND	Operator	Logical operator—performs conjunction
ASC (x$)	Function	Returns ASCII code for 1st character of x$
ATN(x)	Function	Returns the arctangent of x
AUTO	Command	Generates line numbers automatically
BEEP	Statement	Beeps through the speaker
BLOAD f$	Command	Loads a memory image file into memory
BSAVE f$,o,l	Command	Saves portions of computer memory to a file
CALL n	Statement	Calls an assembly language subroutine
CDBL(x)	Function	Converts x to a double-precision number
CHAIN [MERGE]f$	Statement	Transfers control and passes variables to another program from current program
CHDIR	Command	Changes current directory
CHR$(x)	Function	Converts ASCII code to its character equivalent
CINT(x)	Function	Converts x to an integer
CIRCLE(x,y),r	Statement	Draws an ellipse centered at x,y, radius r
CLEAR	Command	Sets all numbers to 0 and strings to blank
CLOSE	Statement	Concludes I/O to a device or file
CLS	Statement	Clears the screen
COLOR	Statement	Sets foregr., backgr., border screen color
COM(x) ON|OFF|STOP	Statement	Enables/disables comm. interrupts for x
COMMON v	Statement	Passes variables to a chained program
CONT	Command	Resumes program execution after a break
COS(x)	Function	Returns the trigometric cosine function of x
CSNG(x)	Function	Converts x to a single-precision number
CSRLIN	Variable	Returns the line no. of the cursor
CVD(x$)	Function	Converts 8 byte string to numeric variable
CVI(x$)	Function	Converts 2 byte string to double prec. var.
CVS(x$)	Function	Converts 4 byte string to numeric variable
DATA c	Statement	Stores num. & string constants for READ
DATE$	Var/Stmt	Sets or retrieves the date
DEF FN_____	Statement	Defines and names a user-written function
DEF SEG	Statement	Defines the current "segment" of storage
DEF USRn	Statement	Spec. memory location of subr. n for USR
DEFDBL	Statement	Declares double-precision variable type
DEFINT	Statement	Declares integer variable type
DEFSNG	Statement	Declares single-precision variable type
DEFSTR	Statement	Declares string variable type
DELETE	Command	Deletes program lines
DIM v(x)|v$(x)	Statement	Specifies maximum values for array subscripts
DRAW x$	Statement	Draws an object as specified by x$
EDIT line	Command	Displays a line for editing
ELSE	Operator	Specifies alt. expression for IF THEN test
END	Statement	Terminates program execution, closes files
ENVIRON p$	Statement	Mod. parameters in BASIC's environ. table
ENVIRON$(x)	Function	Read par. in BASIC's environment table
EOF(x)	Function	Indicates an end of file condition on file x
EQV	Operator	Logical operator—performs equivalence
ERASE	Statement	Eliminates arrays from a program
ERDEV	Variable	DOS interrupt 24 error code variable
ERDEV$	Variable	DOS error device name
ERL	Variable	Returns a BASIC line no. for error
ERR	Variable	Returns a BASIC error code
ERROR n	Statement	Simulates a BASIC error n or allows definition
EXP(x)	Function	Calculates exponential to xth power
FIELD #f,width AS s$	Statement	Allots space for variable in file buffer
FILES	Command	Displays the names of files on a disk
FIX(x)	Function	Truncates x to an integer
FN_____	Function	Calls a user-defined function
FOR v=x TO y	Statement	Begins a series of instructions in a loop
FRE(x)|(x$)	Function	Returns no. of bytes in memory not used
GET #f (files)	Statement	Reads a record from a random file
GET (graphics) (x1,y1) -(x2,y2),a	Statement	Reads points from an area of the screen
GOSUB line	Statement	Branches to a subroutine
GOTO line	Statement	Branches unconditionally to specified line no.
HEX$(x)	Function	Returns the hexadecimal rep. of x
IF x|x$ GOTO|THEN	Statement	Makes a decision regarding program flow
IMP	Operator	Logical operator—performs implication
INKEY$	Variable	Reads a char. from the keyboard if available
INP(x)	Function	Returns the byte read from port x
INPUT x|x$	Statement	Receives input from the keyboard
INPUT#fx|x$	Statement	Reads data from a sequential device or file
INPUT$(x,n)	Function	Returns x char. read from keyb. or file n
INSTR(z,y$,x$)	Function	Searches for x$ within y$ starting at z
INT(x)	Function	Rtns. largest integer less than or equal to x
INTER$	Reserved	International character set
IOCTL #f,x$	Statement	Send control string to device driver
IOCTL$(#f)	Function	Read control data string from dev. driver
KEY ON|OFF|LIST|n,x$	Statement	Sets or displays soft keys or key trapping
KEY(n) ON|OFF|STOP	Statement	Activates / deactivates trapping for key n
KILL x$	Command	Deletes a file from disk
LEFT$(y$,x)	Function	Returns the leftmost x characters of y$
LEN(x$)	Function	Returns the number of characters in x$
LET x=y|x$=y$	Statement	Assigns value of an expression to a variable
LINE	Statement	Draws a line or box on the screen
LINE INPUT	Statement	Reads entire line from keyboard
LINE INPUT #f,x$	Statement	Reads entire line from sequential file
LIST	Command	Lists the prog. cur. in memory to screen
LLIST	Command	Lists the prog. cur. in memory to printer
LOAD f$	Command	Loads a program into memory
LOC(x)	Function	Returns the current position in the file x
LOCATE	Statement	Positions the cursor on the active screen
LOF(x)	Function	Returns the no. of bytes alloted to file x
LOG(x)	Function	Returns the natural logarithm of x
LPOS(x)	Function	Returns pos. of print head for printer x
LPRINT x|x$	Statement	Prints data on the printer
LPRINT USING x$; y\|y$	Statement	Prints on the printer using the format x$
LSET f$=x$	Statement	Moves data into a rdm. file buffer bef. PUT
MERGE	Command	Merges lines from ASCII file to current program
MID$(x$,y,z)	Stmt/Func	Returns part of string x$ starting at y for z

BASIC 3.10 Commands

BASIC	TYPE	DESCRIPTION
MKDIR p$	Command	Creates a new directory
MKD$	Function	Converts double-precision value to string
MKI$	Function	Converts integer value to string
MKS$	Function	Converts single-precision value to string
MOD	Operator	Performs modulo arithmetic
MOTOR	Statement	Turns the cassette player on and off
NAME f$ AS g$	Command	Changes the name of a diskette file
NEW	Command	Deletes the program currently in memory
NEXT	Statement	Ends a series of instructions in a loop
NOT	Operator	Logical operator—performs complement
OCT$(x)	Function	Returns a string for the octal value of x
ON COM(x) GOSUB line	Statement	Sets an asynch port no. for BASIC to trap
ON ERROR GOTO line	Statement	Enables error trapping and specifies line
ON x GOSUB line	Statement	Branches to one of several subroutines
ON x GOTO line	Statement	Branches to one of several line numbers
ON KEY(x) GOSUB line	Statement	Sets line BASIC will trap when key x pressed
ON PEN GOSUB line	Statement	Sets up line no. to transfer to on light pen
ON PLAY(x) GOSUB line	Statement	Plays cont. background music for x notes
ON STRIG(x) GOSUB line	Statement	Sets up joystick for BASIC to trap
ON TIMER(x) GOSUB line	Statement	Branch to subroutine after x seconds
OPEN f$ FOR mode1 AS #f	Statement	Allows I/O to a file or device
OPEN mode2,#f,f$	Statement	Allows I/O to a file or device
OPEN "COMn:..." AS #f	Statement	Sets Async port and opens a comm. file
OPTION BASE x	Statement	Declares the minimum value of an array subs.
OR	Operator	Logical operator—performs disjunction
OUT y,x	Statement	Sends value x to port y
PAINT(x,y)	Statement	Fills screen or pattern area with a color
PEEK(x)	Function	Returns a byte read from a memory pos.
PEN	Function	Reads the light pen
PEN ON|OFF|STOP	Statement	Enables/disables light pen
PLAY ON|OFF|STOP	Statement	Enables/disables background music
PLAY x$	Statement	Plays music as specified by string x$
PLAY(x)	Function	Returns number of notes in music buffer
PMAP(x,n)	Function	Maps between physical and world coordinates
POINT(x,y)	Function	Returns attribute of point x,y on the screen
POINT(n)	Function	Returns value of current screen point x,y
POKE y,x	Statement	Writes a value x to memory location y
POS(x)	Function	Returns the current cursor column position
PRESET(x,y)	Statement	Draws a point on the screen at x,y background
PRINT x|x$	Statement	Displays data on the screen
PRINT USING x$; y\|y$	Statement	Displays data using the format x$
PRINT#f,x|x$	Statement	Writes data seq. to a file
PRINT# USING x$; y\|y$	Statement	Writes data seq. to a file using format
PSET(x,y)	Statement	Draws a point on the screen at position x,y
PUT #f	Statement	Writes a random record to a file
PUT(x,y),a	Statement	Writes image on screen starting at x,y
RANDOMIZE x|TIMER	Statement	Reseeds the random no. generator with x
READ x|x$	Statement	Reads values from DATA statements into variables
REM	Statement	Inserts an explanatory remark in a program
RENUM	Command	Renumbers the program lines
RESET	Command	Closes all disk files and clears the buffers
RESTORE x	Statement	Allows data state. to be reread from line x
RESUME|line|NEXT	Statement	Continues prog. execution after an error
RETURN	Statement	Returns control from a subroutine
RIGHT$(y$,x)	Function	Returns the rightmost x char. of string y$
RMDIR	Command	Removes a directory
RND(x)	Function	Returns a random no. between 0 and 1 on x
RSET	Statement	Moves data into random file buffer before PUT
RUN	Command	Begins execution of a program
SAVE	Command	Saves a BASIC prog. on disk or cassette
SCREEN(x,y)	Function	Returns ASCII code for the character at x,y
SCREEN c$	Statement	Sets the screen attributes
SGN(x)	Function	Returns the sign of x
SHELL	Statement	Execute prog. or BATch file within BASIC
SIN(x)	Function	Calculates the trig. sine function of x
SOUND f,d	Statement	Generates sound through the speaker
SPACE$(x)	Function	Returns a string of x spaces
SPC(x)	Function	Skips x spaces in a PRINT statement
SQR(x)	Function	Returns the square root of x
STEP	Operator	Specifies the increment size of FOR state.
STICK(n)	Function	Returns the x and y coord. of joystick n
STOP	Statement	Stops the program (RESUME continues)
STR$(x)	Function	Returns a string rep. of the value x
STRIG (n)	Function	Returns the status of the joystick button n
STRIG ON|OFF	Statement	Enables/disables joystick button trapping
STRIG(x) ON|OFF|STOP	Statement	Enables/disables trapping joystick buttons
STRING$(n,m\|x$)	Function	Returns a string of n characters
SWAP____ , ____	Statement	Exchanges two variables
SYSTEM	Command	Exits BASIC and returns to DOS
TAB(x)	Function	Tabs to position x in PRINT instruction
TAN(x)	Function	Returns the trigonometric tangent of x
THEN	Operator	Allows for expression based on IF test
TIME$	Var/Stmt	Sets or retrieves the current time
TIMER ON|OFF	Statement	Enables or disables elapsed time trapping
TIMER	Function	Returns seconds since midnight or reset
TO	Operator	Sets the ending value in a FOR NEXT loop
TROFF	Command	Ends trace of execution of program statements
TRON	Command	Starts trace of execution of program statements
USING	Operator	Des. format used in PRINT and PRINT#
USR(n) (arg)	Function	Calls assembly lang. subr. n with arg
VAL(x$)	Function	Returns the numerical value of string x$
VARPTR(V)	Function	Returns the memory address of the variable
VARPTR$(V)	Function	Returns the char. form of a memory address
VIEW|VIEW SCREEN	Statement	Defines a WINDOW subset of screen
WAIT x,a	Statement	Suspends prog. execution until port x input
WEND	Statement	Ends a WHILE loop
WHILE x	Statement	Begins a loop while expression x is true
WIDTH s|d,s|#f,s	Statement	Sets output line width
WINDOW	Statement	Redefines coordinates of viewport
WRITE x|x$	Statement	Outputs data to the screen
WRITE #f,x|x$	Statement	Writes data to a sequential file
XOR	Operator	Logical operator—exclusive or

Media Check (Function 1)

The Media Check function:

- Reads the media descriptor byte.
- Tests the media.
- Sets the status word.
- Sets the media return code.

The media return code can be -1, indicating a change; 0, for unknown result; or +1, indicating no change. In DOS versions 3.00 and later, if the return byte has been set to -1, and the device driver sets bit 11 of the device header attribute on, the driver must set the pointer to the previous volume ID. DOS then determines if that change is an error and returns an invalid disk change error on behalf of the device.

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 01H
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS
+13	1 byte	Media descriptor byte from DOS
+14	1 byte	Media return code
+15	2 bytes	Pointer to previous volume ID (DOS 3.00 and DOS 3.10)

Figure 2. Media Check (Function 1)

Build BIOS Parameter Block (Function 2)

The Build BIOS Parameter Block function:

- Checks the device to find the descriptor byte.
- Finds the matching BIOS parameter block.
- Sets the direct pointer to the BIOS parameter block.

This function causes the device driver to read the boot sector of the media for the media description.

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 02H
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS
+13	1 byte	Media descriptor byte from DOS
+14	2 bytes	Transfer address (offset)
+16	2 bytes	Transfer address (segment)
+18	2 bytes	Direct pointer to BIOS parameter block (offset)
+20	2 bytes	Direct pointer to BIOS parameter block (segment)

Figure 3. Build BIOS Parameter Block (Function 2)

Input or Output (Functions 3, 4, 8, 9, 12)

The INPUT or OUTPUT function:

- Reads the sector byte count.
- Performs the requested function.
- Sets the actual count of sectors or bytes.
- Sets the status word.

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 03H, 04H, 08H, 09H, 0CH
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS
+13	1 byte	Media descriptor byte from DOS
+14	2 bytes	Transfer address (offset)
+16	2 bytes	Transfer address (segment)
+18	2 bytes	Byte count (character device drivers) Sector count (block device drivers)
+20	2 bytes	Starting sector number (block devices only)
+22	4 bytes	Pointer to volume-ID if 0FH error occurs (DOS 3.00 and DOS 3.10)

Figure 4. Input or Output (Functions 3, 4, 8, 9, 12)

Nondestructive Input No Wait (Function 5)

The Nondestructive Input No Wait function:

- Provides a one-character look-ahead (character devices only).
- Sets the status word.
- Does not alter the input buffer.

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 05H
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS
+13	1 byte	Byte read from device)

Figure 5. Nondestructive Input No Wait (Function 5)

Input or Output Status (Functions 6, 10)

The Input or Output Status function:

- Determines the status of the input or output device.
- Sets the status word.

There is no data associated with this function.

Input or Output Flush (Functions 7, 11)

The Input or Output Flush function:

- Clears the input or output buffer.
- Sets the status word.

There is no data for this function.

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 06H, or 0AH
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS

Figure 6. Input or Output Status (Functions 6, 10)

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 07H, or 0BH
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS

Figure 7. Input or Output Flush (Functions 7, 11)

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 0DH, or 0EH
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS

Figure 8. OPEN or CLOSE (Functions 13 & 14) (DOS 3.00 and DOS 3.10 only)

DISPLACE	LENGTH	DESCRIPTION
+00	1 byte	Length of header + data
+01	1 byte	Unit code (block devices only)
+02	1 byte	Function code 0FH
+03	2 bytes	Status code (set by device)
+05	8 bytes	Reserved for DOS

Figure 9. Removable Media (Function 15) (DOS 3.00 and DOS 3.10 only)

Open or Close (Functions 13, 14)

The Open or Close function is for setting a device before starting a task.

Removable Media (Function 15)

The Removable Media function sets the busy bit (9) of the status word to 1 if the media is non-removable.

Installing Device Drivers

Your device driver program must be available when you start or reboot your system. DOS looks for references to device drivers in its CONFIG.SYS file. This file must be in the root directory of drive A or C, and must contain the line:

DEVICE=[d:] [path] filename.ext

where

[d:] is the name of the drive where the device driver program is located.

[path] is the path where the device driver program is located.

filename.ext is the name and extension of the device driver program.

When DOS is loaded, it looks for all the device drive programs specified in CONFIG.SYS and makes them part of the operating system. If a character device driver name (listed in the device header) has the same name as an existing device driver, such as LPT1, DOS will use the new device driver in place of the old one. In the case of a block device driver, DOS will count the number of units, place that amount in the first byte of the driver name, and assign the appropriate number of device letters to the driver.

Using Device Drivers

Using a new device driver is relatively easy. You treat it exactly like other DOS devices: CON, LPT1, COM1, A, B, C, etc. If the device driver is written properly, you should be able to issue commands to the new device, or open it for input and output, just like any other DOS device.

The device driver should make the input and output from the device transparent to the user. The only exception is in a program, such as BASIC, that does not use standard input or output. However, BASIC allows you to open the device as a file, send control information to it through the IOCTL statement, and read control information from it with the IOCTL$ function.

For a character device driver, the system uses the name built into the device driver. As stated before, if the device name is the same as an existing one, DOS uses the new driver in place of the old one. This can be useful when you use an asynchronous device driver to make a "dumb" terminal the console. (A dumb terminal is one with no built-in processing capabilities, in contrast to a "smart" terminal, which has some intelligence.) For example, creating an asynchronous driver and calling it CON will make the system use a remote terminal in place of your display and keyboard.

For a block device driver, DOS assigns that device the next available letter as the device name. An instance of this is where you use a virtual disk device driver with system memory. DOS defines it as drive C, D or E depending on how many drives and adapter cards are installed on your system.

Communicating with Device Drivers

You use a device driver in order to convey commands to a "smart" physical device, to make a "dumb" hardware device look "smart" to the system, or to pass non-system information to the device. Communication can take two forms: byte stream communication or control channel communication.

In byte stream communication, the driver must recognize some kind of escape character in the byte stream and use the characters that follow the escape as control characters. For example, byte stream communication is used when positioning the cursor with the ANSI.SYS device driver.

Control channel communication requires that the driver recognize a hardware control channel. In control channel communication, you:

- Build the message.
- Build the device driver name.
- Create the handle (file control block).
- Send a message to the handle.
- Close the handle.

Control channel communication is used, for example, when telling a printer to indent five spaces after each line feed.

The following is an assembly language program that performs control channel communication:

```
             —                       ;
             —                       ;
             JMP     NEXT            ;
MSG          DB      'OK',13,10      ; BUILD THE MESSAGE
FILE         DB      'CHRDVR',0      ; BUILD THE DEVICE NAME
                                     ;
NEXT         PUSH    DS              ; save DS register
             MOV     AX,CS           ; set DS register to
             MOV     DS,AX           ;    equal the CS register
             PUSH    BX              ; save the other registers
             PUSH    CX              ;
             PUSH    DX              ;
                                     ;
                                     ; CREATE THE HANDLE
             MOV     AH,3CH          ; function 3CH; create file handle
             MOV     CX,0            ; select the file attribute
             LEA     DX,FILE         ; DS:DX point to the file/device
                                     ;    name
             INT     21H             ; execute DOS function call
             JC      EXIT            ; exit on error
                                     ;
                                     ; SEND THE MESSAGE
             MOV     BX,AX           ; move file handle to BX
             MOV     AH,44H          ; function 44H; IO-CNTL
             MOV     AL,3            ;    write IO-CNT message
             MOV     CX,4            ;    length of message
             LEA     DX,MSG          ; DS:DX points to message
             INT     21H             ; execute DOS function call
                                     ;
                                     ; CLOSE THE HANDLE
             MOV     AH,3EH          ; function 3EH; close handle
                                     ;    (BX still contains the handle)
             INT     21H             ; execute DOS function call
                                     ;
EXIT:        POP     DX              ; restore the registers
             POP     CX              ;
             POP     BX              ;
             POP     DS              ;
CONT:        —                       ; and continue . . .
             —
```

Figure 10. Control Channel Communication Program

TopView Questions and Answers
(Part 2)

Compiled by the TopView Development Team
IBM Corporation

Editor's Note: This is the second part of an article covering TopView questions and answers. Part 1, published last month, discussed the General and Multitasking categories.

The following questions and answers are intended to assist developers of IBM Personal Computer applications in evaluating TopView and the TopView Programmer's ToolKit.

Questions and answers fall into these categories:

- Windowing
- Data Exchange
- Program Information File
- TopView Programming Conventions
- Language Interface Support
- Mouse Support
- Device Driver Support
- Problem Determination

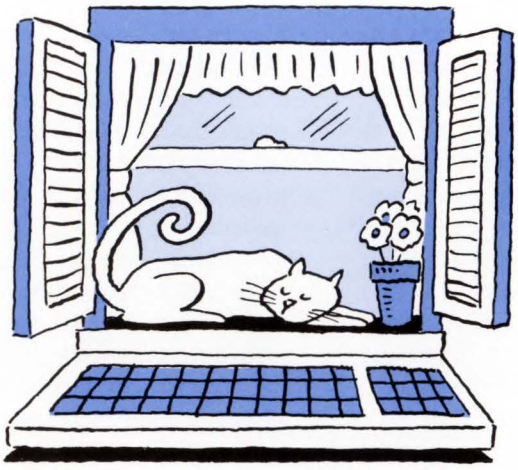
Windowing

Q42: Can an application have more than one window?

A: An application can have as many windows as it needs. There is a limit of 255 windows in TopView; however, some of these are already used by TopView. The major constraint on the number of windows in a system is the amount of system memory available.

Q43: Does the application determine the initial size of a window? Can it be a different size than full screen (25 x 80)? What are the minimum and maximum sizes of a window?

A: Yes, the application determines the initial (and maximum size) of each window when the window object is created. The maximum size of a window is 127 x 127. The minimum size is 0 x 0. However, the maximum size window that can be created by the Window Design Aid is 25 x 80.



Q44: Can an application change the size of a window or move the window on the screen?

A: Yes. Refer to the *TopView Programmer's ToolKit* reference book, Chapter 4, "Defining Window Data Streams."

Q45: Can the same window be used for both input and output?

A: Yes, TopView windows can be read from as well as written to.

Q46: Is an application required to have a window?

A: Yes. However, the window may have a size of 0 x 0 and be hidden from view.

Q47: Can an application have a window that can't be viewed by the user?

A: Yes, an application can hide a window when the user is not supposed to see it.

Q48: Does TopView support color for windows? What colors does TopView use? Can the user change the colors?

A: Yes. TopView supports both the color text and color graphics modes on the Color/Graphics Monitor Adapter. TopView has conventions for color usage and defaults for both TopView usage and application usage. Application developers can use either the pre-defined TopView colors or colors of their own choosing. Colors used for TopView functions can be changed to black and white if preferred, or, if the user has a black and

white monitor, connected to the Color/ Graphics Monitor Adapter. Colors used by an application can be changed by the user if the application provides the facility.

Q49: Can applications share a single window?
A: Yes. It is possible but not recommended because of the difficulty of coordinating updates to the window.

Q50: Does TopView support multiple displays at the same time?
A: TopView allows users with both a monochrome and a color adapter to run applications on both displays. However, only one display is active at a time. The current foreground (interactive) application determines which display is active.

Q51: Can an application update windows on different displays at the same time?
A: No. An application can use only one display at a time, but it can change video modes.

Q52: Can an application cause the screen to be redrawn, or is that under TopView's control?
A: The scheduling of the physical redrawing of the screen is under TopView's control. However, an application can request that any of its windows be redrawn at any time. The requesting application is suspended until the redraw has occurred.

Q53: What input modes does TopView support (full screen, fielded data, etc.)?
A: TopView supports a full-screen input mode that allows a complete set of fields to be returned to the application at one time. It also provides character editing and cursor positioning functions. A keystroke input mode allows the application to interpret and process each keystroke entered by the user.

Q54: Does TopView support field validation for keyboard input data?
A: No. TopView provides several editing functions, such as uppercase conversion and automatic clearing of default data. However, no checking is performed on data entered in a field. TopView does provide a mechanism by which a special routine in the application can validate keystrokes.

Q55: What is the Window Design Aid?
A: The Window Design Aid is an interactive tool for designing windows for a TopView application. A window created by the Window Design Aid is stored in a disk file as a sequence of data stream codes called a panel.

Q56: How does an application use a panel object?
A: When an application applies a panel to TopView, the data stream codes are used to create and/or define a window.

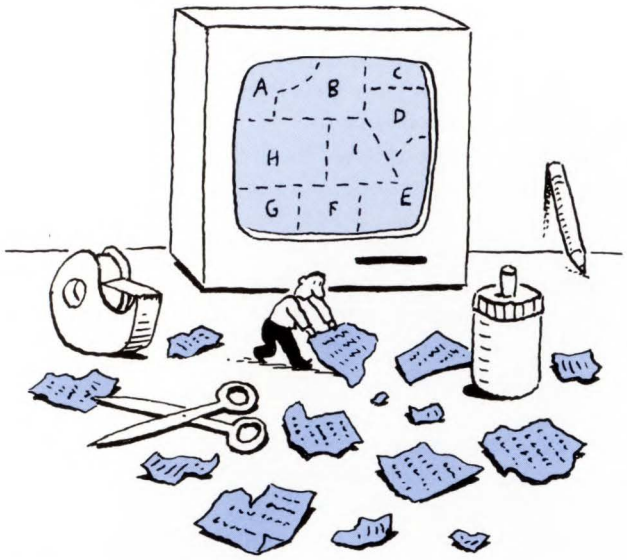
Q57: How does an application access panels created by the Window Design Aid?
A: TopView provides facilities for accessing panels that reside on disk or in RAM memory. A ToolKit utility merges groups of panels into a library of panels so they can be accessed more readily from your program. The TopView Programmer's ToolKit also contains a utility for converting panels or a panel library to an object module that can be linked with an application.

Data Exchange

Q58: What data exchange function does TopView support?
A: TopView supports the CUT, COPY and PASTE of ASCII data taken from or inserted into a window. However, TopView provides only the COPY function directly. The CUT and PASTE functions must be implemented by each application as appropriate. Existing applications can implement these functions by providing appropriate filter table entries that translate the selection of a CUT or PASTE option into the appropriate keyboard commands for the application.

For example, a CUT might be translated into a BLOCK DELETE application command. When the user selects CUT, the marked data would be copied to a TopView buffer and the BLOCK DELETE command would be passed to the application just as if that command had been entered from the keyboard.

New applications written using the TopView Applications Programming Interface can implement the CUT and PASTE functions without requiring filter tables.



Q59: What is the syntax of the filter table entries for CUT, COPY and PASTE functions?

A: Refer to the *TopView Programmer's ToolKit* reference book, Chapter 7, "Creating a Filter Table".

Q60: How does TopView know how to manipulate the data in the Application windows in order to perform the data exchange functions?

A: TopView does not manipulate the data in an application's window or in an application's file. These functions are the responsibility of each application. TopView provides a facility to initiate the function in a standard way and a facility to store and transfer data between applications.

Program Information File

Q61: What information is required by TopView so that an Application can run with TopView?

A: TopView must have the following information in order to run an application:

- Program title
- Full pathname of the program's start-up file (*.COM, *.EXE, etc.)
- Data files location (default drive and directory)
- Memory requirements (minimum, maximum and system)
- Screen type (e.g., 25 x 80 color, etc.)
- Number of screen pages
- Range of software interrupt vectors used (modified)
- Window size and offset
- Shared program information
- Program behavior characteristics

Note: Refer to the *TopView Programmer's ToolKit* reference book, Chapter 7, "TopView Program Information Files," for a detailed description of each of these items.

Q62: Where should the application's Program Information File be located?

A: The Program Information File must be located on the same drive and directory as the application itself.

Q63: Does a user have to develop the Program Information File for every application?

A: Program Information Files have been developed for some existing applications that are compatible with TopView. These files are shipped on the TopView diskettes. However, developers of new applications that run with TopView can develop a Program Information File as part of their development process and ship it on the diskette that contains their application.

It also is possible to run an existing application that does not have a Program Information File. When the user adds (installs) the program, TopView searches the location (disk and directory) for a file named 'program name.PIF'. If no file with that name is found, the user is prompted for some of the information. The user has to enter only the program title, the location and name of the start-up file, and the memory requirements. TopView gives default values to the remaining information. The user can modify the program information later if necessary.

Q64: Is the user expected to remember and specify all the information in the Program Information File each time the program is used?

A: No. When the user adds (installs) a program, TopView creates a record of the information in a master file used by TopView. There is one record in the file for each application that has been added. This record must exist before a program can be started, but it need be added only once.

Q65: When a Program Information File is not supplied on the application diskette, does TopView supply any defaults?

A: Yes. TopView supplies defaults that will (in most cases) allow the application to run. However, the default values may need to be modified in order to make the best use of TopView features.

Q66: What happens if the program information for an application is incorrect?

A: In most cases, an error message will be displayed when the user tries to start the program (e.g., "File not found" or "Not enough memory"). In other cases, the results can be unpredictable.

TopView Programming Conventions

Q67: What interrupts does TopView use?

A: Interrupts 50H through 57H; 15H (AH = 10, AH = 11, AH = 12).

Q68: Can an application use the TopView interrupts too?

A: Applications can use INT 15H (AH = 10, AH = 12) to make use of TopView-specific facilities, but should not use the other interrupts.

Q69: What is the TopView call/return convention?

A: Refer to "The TopView Subroutine Call" and "The TopView Object/Message Call" in Chapter 2 of the *TopView Programmer's ToolKit* reference book.

Q70: Does TopView support shared re-entrant code?

A: Yes, through the use of a shared program and data. See the *TopView Programmer's ToolKit* reference book, Chapter 8, "Creating Compatible Applications," for a discussion about shared programs.

Q71: What are the TopView subroutine calls?

A: See answer to Q69.

Q72: Is there a recommended way to use the printer so that my application can share the printer resource?

A: The recommended method for using the printer is to OPEN the printer (e.g., LPT1:), WRITE to the printer using DOS or BIOS functions, and then CLOSE the printer. The printer remains in control of your application. However, TopView provides a mechanism by which the user can give control of the printer to another application that requests it.

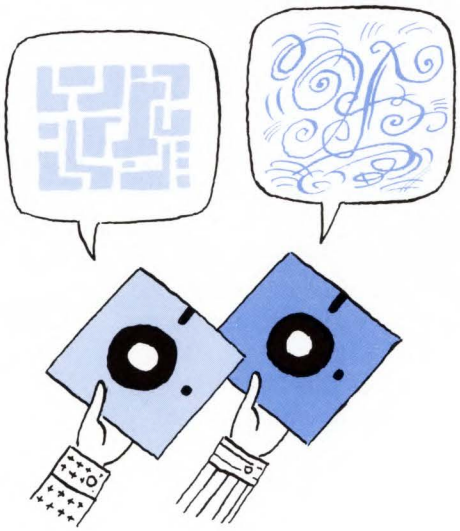
Q73: Does TopView reserve the use of any keys?

A: Although TopView uses certain keys, it does not preclude an application from using any key. For example, the Alt key is used by many applications in conjunction with another key, and is used by TopView to display the main menu. If the Alt key is held down and another key is pressed, the keystroke is passed to the application. If the Alt key is pressed and released, TopView displays its main menu.

Language Interface Support

Q74: What languages are currently supported?

A: TopView currently supports an interface to IBM 8088 assembly language programs. It also provides a sample interface to IBM Pascal. However, programs that do not use the TopView API can be written in any language as long as they do not violate the TopView well-behaved program guidelines.



Q75: Are there plans to support any other languages?

A: Not at this time. However, using the existing interface as an example, an application developer could write an interface to another language. The source code for the TopView interface to Pascal is provided on the TopView Programmer's ToolKit diskette to help facilitate this translation.

Q76: What functions do the language interface routines provide the developer?

A: The interface routines provide full access to the TopView subroutine calls (INT 15, AH = 10) and object message calls (INT 15, AH = 12). See the *TopView Programmer's ToolKit* reference book for a complete description.

Mouse Support

Q77: Is a mouse required in order to use TopView?

A: No. A mouse is optional for use with TopView. The keyboard can be used as the TopView pointing device.

Q78: Must all applications support a mouse pointing device in order to run with TopView?

A: No. Applications do not have to provide support for a mouse.

Q79: What mouse devices are currently supported?

A: TopView currently supports the following mouse devices:

- Microsoft Mouse for IBM Personal Computers™, model number 037-099 (parallel interface)
- Microsoft Mouse for IBM Personal Computers™, model number 039-099 (serial interface)
- PC Mouse™ by Mouse Systems, part number 900120-214 (serial interface)
- Visi On Mouse™ by VisiCorp, part number 69910-1011 (serial interface)
- Other pointing devices that have not been tested by IBM may work with TopView.

Q80: Is there a special way to set up a mouse to work with TopView?

A: Most pointing devices connect to an RS-232 serial communications port. These devices may be connected to either COM1: or COM2:. Take care to ensure that there is no conflict between the pointing device and any communications programs over the use of a particular communications port. When operating with two RS-232 attachments, it is recommended that any serial mouse be attached to COM2:. In addition, the software drivers required for all of the mouse devices listed above are supplied on the TopView diskette. It is not necessary to install any software that comes with the mouse.

The Microsoft Mouse (parallel interface) connects to an interface card that must be installed in an expansion slot. This card has a small jumper which selects the interrupt level that will be generated by the interface card. The setting of this jumper can be very important, since the interrupts generated by the interface card must not be at the same level as any other interrupts generated in the system.

Q81: Can other pointing devices that are currently not supported by TopView be used?

A: Yes. Other types of pointing devices that satisfy the following conditions may be used:

- A standard TopView pointing device driver has been provided for the particular device.
- The device is capable of presenting either absolute position information or relative position information that can be tracked and converted to absolute position information.
- The device has three independent "buttons" or their equivalent. For example, devices with two pushbuttons often map the third button to the simultaneous pressing of both buttons.

Device Driver Support

Q82: Can device drivers supplied with TopView be modified to support other devices or adapter cards?

A: No. The source code is not provided.

Q83: Can a device driver be written that will run with TopView?

A: Yes. See the *TopView Programmer's ToolKit* reference book.

Q84: Can a device driver be written that will run both with and without TopView?

A: Any device driver (except a pointer device driver) written according to DOS 2.00, 2.10, 3.00 or 3.10 device driver format will run with or without TopView. However, device drivers that use the CON: device, perform keyboard I/O or are not well behaved will not run with TopView. For example, ANSI.SYS is not supported.

Problem Determination

Q85: Does TopView provide any debug aids for determining what caused a problem?

A: No. TopView does not provide a debug aid. However, the IBM Professional Debug Facility can be used with TopView; it was used during TopView development.

Decimal Numbers In BASIC

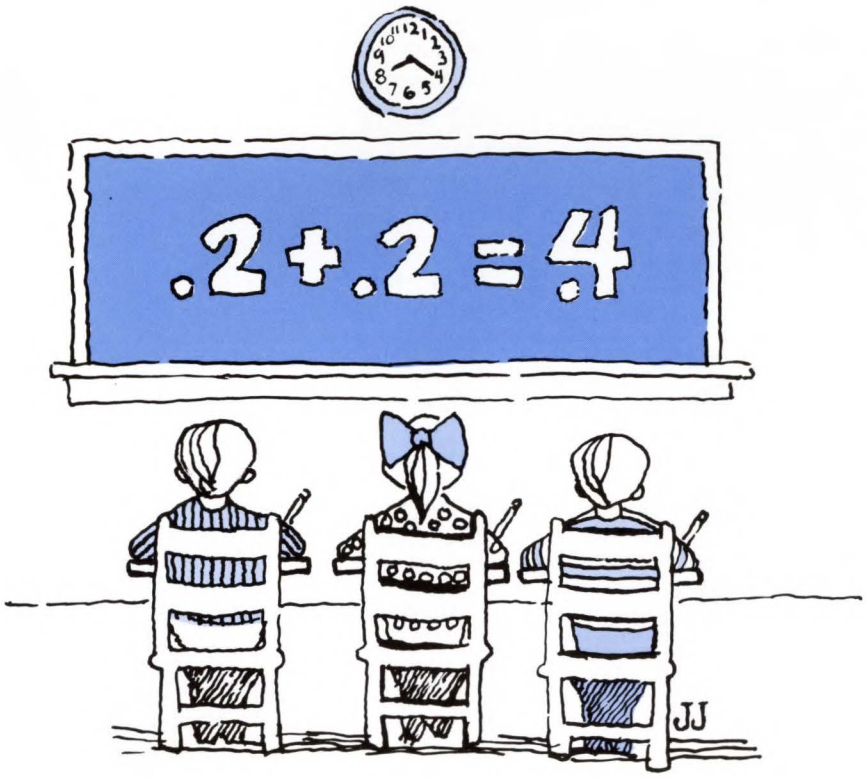
Scott Wellman
IBM Corporation

In IBM Personal Computer BASIC, a decimal number sometimes cannot be stored directly in binary (internal machine) form. A decimal number will first be converted to floating point notation and then into binary form.

Just as 10 divided by 3 does not yield a whole number (transcendental or non-repeating), a number such as .09 cannot be represented as a binary number. The computer stores it as .09000001 when using single-precision floating point variables and prints it as 9.000001E-02.

One way to correct the printed number is to print with the PRINT USING statement, which rounds off the number to a more reasonable display. Instead of the statement PRINT .09, use PRINT USING "##.##"; .09, which prints as 0.09. (For more information about PRINT USING, see the *BASIC Reference* manual).

The PRINT USING statement solves how the number prints out, but not the way the number is stored in the computer. Normally, however, you wouldn't care how the computer stores numbers, unless you write extensive scientific programs. But if you are writing even a simple math program, such as a checkbook balancer, a small error repeated many times can result in a large error. For example, run the following program and notice what answer is printed.



Problem: Subtract .09 from 100,000 a thousand times.

Correct answer: 99,910

```
10 TOTAL = 100000: NBR
   = .09 'Assign numbers
20 FOR I = 1 TO 1000
   'Loop 1000 times
30 TOTAL = TOTAL -
   NBR 'Subtract
40 NEXT I
50 PRINT TOTAL 'Print
   answer
```

Printed answer: 99,906.25

(I'd love to have $100,000 in my checking account, but I'm not so sure I'd want to write a thousand checks for 9 cents!)

The easiest approach to improving the accuracy of the number in storage is to use double-precision, floating point numbers. In the program above, instead of TOTAL and NBR, use TOTAL# and NBR# (or see DEFtype in the *BASIC Reference* manual). The program-generated answer will be 99909.9999....., which is more accurate, but not precisely correct, because there is still no way to keep .09 as .09 in binary form.

By replacing the PRINT statement with PRINT USING "######.##"; TOTAL#, the result-

ing answer improves to 99910.00. If you are balancing your checkbook, this technique is probably close enough so that you will never notice a discrepancy. Larger decimal numbers (e.g., 8.9) are similarly affected.

We still haven't determined how to process fractional numbers precisely over many iterations. BASIC has several built-in functions (INT, FIX, CINT, CSNG and CDBL) that strip off decimal places and return only whole numbers, or round up and down. Each function has special properties that affect results, usually by rounding. For example, INT will work only if the number is between 32767 and -32768, and it drops the decimal value, e.g., 8.9 becomes 8, and -8.9 becomes -9.

The best approach is to store the number in the computer as a whole number. Instead of 8.9, use 89. However, your program must keep track of where the decimal point is supposed to be. When you wish to display or print the value, you may have to divide the results by some power of 10 (e.g. TOTAL#/100).

For applications that use even smaller decimal numbers, a larger divisor is required. A quick rule is: the number of decimal places you require should equal the number of zeros in the divisor. For example, in an inventory program, 8 nails cost 1 cent, so 1 nail costs .125 cent = .00125 dollars = 125/100000. Notice 5 decimal places in the number and 5 zeros in the divisor.

Now that you have fooled the computer into keeping the number correctly, you certainly don't want to fool the user, too. You want the user to be able to enter 8.9 rather than 89. To do this, you must add program code that accommodates the correct input.

Again, use double-precision numbers to be as close to correct as possible. A few lines of code might be:

```
10 INPUT "Enter amount of
   check"; CHK#
20 TOTAL# = TOTAL# -
   (CHK# * 100)
```

These two lines merely multiply the number entered by 100. The number 8.9 x 100 is stored as 889.999961.... Since this is close, but not the correct value, round the number before you multiply it:

```
30 TOTAL# = TOTAL# -
   ((CHK# + .0001) * 100)
```

Purists will insist you should correct the sign of the value of CHK#, e.g., ((-8.9 + .0001) * 100) = -889.989866.... The code then appears:

```
(CHK# + (.0001 *
SGN(CHK#))) * 100
```

The value of CHK# is 890.00996...., again close, but not exact. Eliminating the decimal places will make the number correct. The INT function may be used if the number is between 32767 and -32768. I prefer to use the FIX function to strip off the decimals.

```
20 TOTAL# = TOTAL# -
   FIX((CHK# + (.0001 *
   SGN(CHK#))) * 100)
```

Great! You now have the correct answer and can write a thousand checks for 9 cents each and maintain a correct balance.

This program code is cumbersome, however, and if you use the same code in several places, it will mean a lot of careful coding (with probable errors). It is much more reliable to define a function that you can use anywhere in the program. I have chosen to call the function DF for Decimal Fix. When passed the value of CHK#, the DF function will correctly convert it to a whole integer. The following code defines the DF function:

```
10 DEF FNDF(CHK#) =
   FIX((CHK# + (.0001 *
   SGN(CHK#))) * 100)
20 TOTAL# = TOTAL# -
   FNDF(CHK#)
```

Now, go back and recode the original problem, add a line of code that will input a value, subtract it 1000 times, and print the results.

```
 9 REM Create the user
   function to fix the input
10 DEF FNDF(CHK#) =
   FIX((CHK# +(.0001 *
   SGN(CHK#))) * 100)
19 REM Assign the begin-
   ning value, * 100 for no
   decimals
20 TOTAL# = 100000 * 100
30 INPUT "Enter a num-
   ber"; NBR# 'Get a test
   number
40 FOR I = 1 TO 1000
   'Loop 1000 times
50 TOTAL# = TOTAL# -
   FNDF(NBR#) 'Subtract
60 NEXT I
70 PRINT USING
   "######.##";
   TOTAL# / 100
   'Print answer
```

The results will be:

```
RUN
Enter a number? .09
  99,910.00
OK
RUN
Enter a number? 8.9
  91,100.00
OK
```

Structured Programming With BASIC's FOR-NEXT Loop

Phil Niehoff
Madison IBM-PC Users' Group

Because BASIC is an unstructured programming language, it is very easy to write sloppy program code. Sloppy code, often called "spaghetti code," is so unstructured that it is usually readable by only the person who wrote it (and then only if reading it five minutes after it was written). However, one can avoid writing spaghetti code programs by disciplining oneself to structure the code and creating program code that a subsequent programmer can read and understand with little effort. Using and structuring the FOR-NEXT loop is one way to improve the readability of BASIC programs.

The FOR-NEXT loop is used to perform repetitive tasks in a program. The concept is simple. Every statement between the FOR and the NEXT statement will be executed a specified number of iterations.

This effect could be accomplished without using a FOR-NEXT loop. For example, if you wanted to add all the numbers between 1 and 10 and print the result, your program might look like the following:

```
10    X=1
20    Y=0
30    Y=Y+X
40    X=X+1
50    IF X>10 THEN PRINT Y : END
60    GOTO 30
```

Line 10 sets the counter (variable X) to 1. X will be incremented by one until it reaches 10. Line 20 sets variable Y, the variable by which the sum will be accumulated to 0. Line 30 adds the value of X to the previous value of Y and saves the result back in Y. Line 40 increments the counter (X) by one. Line 50 determines if the value of X has exceeded 10. If it has, Y, the sum, is printed, and the program stops. If the value of X is 10 or less, line 60 tells the program to go back to line 30 and continue.

The same program can be written using a FOR-NEXT loop as follows:

```
10    Y=0
20    FOR X=1 TO 10
30      Y=Y+X
40    NEXT X
50    PRINT Y
60    END
```

Line 10 sets the value of Y, the variable in which the sum will be accumulated to 0. Line 20 starts the FOR-NEXT loop and will execute all of the statements between it and line 40, the NEXT statement, 10 times (from 1 to 10). Each time through the loop, X will be incremented by one. Line 30 adds the value of X to the previous value of Y and saves the result back in Y. Line 50 prints the value of Y, the sum.

The first example could be called an unstructured program. Notice that the program is relatively harder to understand than the second program. All of the program lines start flush-left. The use of the GOTO statement often indicates poor program planning. The GOTO statement can almost always be avoided in any program.

The second example does not contain any GOTO statements. The X at the end of the NEXT statement in Line 40 is optional. If the variable is not indicated, the computer will match the NEXT statement to the immediately preceding active FOR statement. A FOR-NEXT loop is active when the program has encountered a FOR statement, but has not yet executed the loop the indicated number of times.

Notice also that all of the statements between the FOR and the NEXT statement are indented, which allows the programmer to visually match FOR and NEXT statements easily. Indenting is especially important when FOR-NEXT loops are nested:

```
10    FOR X=1 TO 10
20      PRINT X
30      FOR Y=20 TO 30
40        PRINT X;Y
50      NEXT Y
60    NEXT X
```

In all preceding examples, the FOR loop is incremented by one each time the loop is executed. If you want to count to 100 by 4's, it's not quite as easy.

```
10    FOR COUNT=0 TO 100 STEP 4
30      PRINT COUNT
40    NEXT COUNT
```

This example varies from the previous ones in two respects. First, notice that the program starts at 0. This is called the initial value of the loop. The initial value is required so that the last number printed would be exactly 100 (the terminal value). If we had started at 1, the last number printed would have been 97. The initial value can be set to any value, negative or positive.

Second, the FOR statement contains the word STEP with a number following it. This allows the

program to skip over numbers as it passes through the loop. The 4 is called the step value. When the step value is one, the STEP command may be excluded since the default value is 1. Notice also that the loop variable (in this case COUNT) must conform to the normal naming conventions for BASIC variables.

The step value can be either positive or negative. Below is an example of a FOR-NEXT loop that counts from 100 back to -100 by 3's.

```
10 FOR LOOP=100 TO -100 STEP -3
20 PRINT LOOP
30 NEXT LOOP
```

Unlike the DO loops in FORTRAN that are executed once before the conditions of the loop are tested, the FOR-NEXT loop in BASIC is tested before the statements in the loop are executed. Therefore, in the following BASIC FOR loop:

```
FOR LOOP=10 TO 1
```

none of the statements in the loop would be executed since 10 is greater than one, and without a negative step value, it is impossible to count from 10 to 1.

Also, a FORTRAN DO loop variable cannot be changed inside the loop; however, in BASIC the control variable can change inside the FOR-NEXT loop. This difference can save time, but for the unwary BASIC programmer, it can cause major problems.

Assume that you have five numbers stored in an array, and that you want to write a program that indicates in which element of the array a certain number is stored. The program might appear:

```
 10 DIM ARRAY(5)
 20 ARRAY(1)=7
 30 ARRAY(2)=9
 40 ARRAY(3)=5
 50 ARRAY(4)=9
 60 ARRAY(5)=3
 70 INPUT "Number:";A
 80 FOR X=1 TO 5
 90 IF ARRAY(X)=A THEN PRINT X
100 NEXT X
110 END
```

If you typed in the above program and entered the number 9 when asked for a number, the program would print the numbers 2 and 4 on the monitor. This type of program works well for looking up numbers in a table. The program compares the number typed in with each of the numbers in the array, one at a time. When a match is found, it prints the number of the array element containing the requested number, then continues matching the rest of the elements of the array to determine if there are any further matches.

This is fine, unless you know that a number can appear in the array only once, or you are interested only in the first occurrence of the number, in which case it is not necessary to continue searching after the match is found. If the array is very large (in the hundreds), you might not want the program to continue searching through the array, since it is not only unnecessary but may take a considerable length of time. So you change line 90 to read:

```
90 IF ARRAY(X)=A THEN PRINT X :
   GOTO 110
```

With this line inserted in the program, the number of the element is printed when a match is found, and the program control jumps to line 110, causing the program to stop executing. This code accomplishes the objective, but it uses a GOTO statement. It also leaves the FOR-NEXT loop active. To see this, type GOTO 100 after the program runs. The FOR-NEXT loop starts executing again from where it left off. If too many FOR-NEXT loops are left active, the memory stacks could overflow, and the computer would issue an error message informing you it is out of memory.

To solve this problem, change line 90 to read:

```
90 IF ARRAY(X)=A THEN PRINT X :
   X=5
```

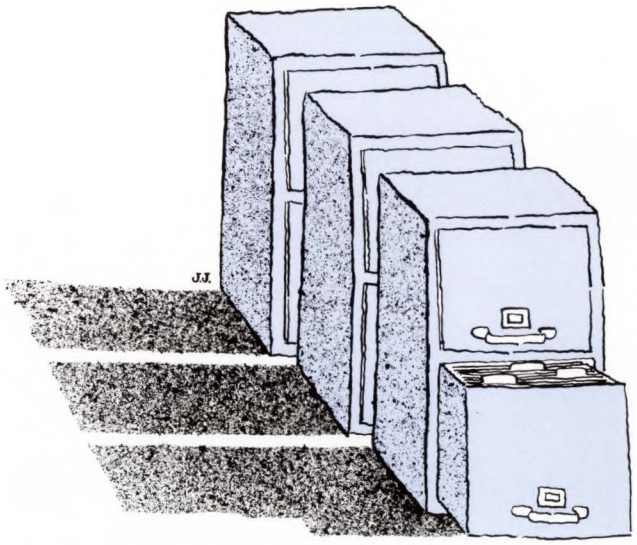
Because BASIC allows you to set the loop variable within the loop, the loop will terminate normally when the NEXT statement is encountered after a match is found. By setting the value of the loop variable to its terminal value, you will fool the loop into thinking that it has gone through the loop for the indicated number of times.

This can be verified by running the program, entering a 9 when prompted to enter a number, then typing GOTO 100 when the program stops running. You should see a message that says "NEXT without FOR in 100." This solution also avoids the GOTO statement.

With a little practice, you should have no trouble using FOR-NEXT loops in your programs; just remember to keep those loops structured with appropriate indenting.

Sequential File Input/Output In BASIC

Vance Jaqua
Rocketdyne PC Users Group



BASIC accesses two types of files: sequential and random access. Sequential files store information in direct sequence, and, when BASIC reads these files, it must start at the top and read through it from the first byte to the end-of-file. To illustrate how BASIC handles sequential files, I will use a program that creates a data file of checks that can then be sorted.

The program must first open the file and name it. It must also instruct the computer whether to receive input from or to write output to this file. The first few lines of your program might look like this:

```
1000 REM PROGRAM TO CREATE DATA FILE
     FOR CHECKS
2000 INPUT "NAME DESIRED FOR FILE";NM$
3000 OPEN "O",#1,NM$
```

Line 3000 opens file #1 as a file that will recieve output ("O") from the program and assigns it the name (NM$) which the user enters when prompted to type a file name. BASIC numbers the file to allow you to have more than one file open at a time, and the program may be reading from one file and writing to another.

Files created using this program are ASCII files that contain information you can read and edit with a text editor. The file needs no special extension. If you desire, you can assign an extension, such as .CHK, to the file name to help you identify the file, but an extension is not necessary.

We will now enter the data from the checkbook.

```
3100 INPUT "INPUT CHECK NUMBER (ZERO
     TO END SESSION):",CHKNO
3110 IF CHKNO=0 THEN 5000
3120 INPUT "CHECK AMOUNT:",AMNT
3140 INPUT "NAME CHECK MADE OUT
     TO:",NAM$
3160 INPUT "DATE NAME OF MONTH, DAY
     NUMBER, YEAR:",MO$,DAY,YR
3180 INPUT "REASON OR PURPOSE OF
     CHECK:",PURP$
3200 INPUT "TYPE FOR TAXES 1)MED, 2)
     TAXES, 3)BUS EXP, 9)MISC:",TN
```

To output this information to the file for storing, use the following format:

```
4000 WRITE#1,CHKNO,AMNT,NAM$
4020 WRITE#1,MO$,DAY,YR,PURP$,TN
4500 GOTO 3100
5000 REM SESSION COMPLETED
5100 CLOSE
```

This program will repeat the questions and prompt you to enter data until you input a zero for CHECK NUMBER to tell the program that you have completed the entry session. The program will then close the file. (Before the file can be used, it must be closed.)

You can now sort this file of checks according to type. For example, you may want a listing of all checks used to pay medical bills. Below is an example of such a sort program:

```
1000 REM PROGRAM TO READ DATA FILE OF
     CHECKS
2000 INPUT "NAME OF FILE TO BE
     REVIEWED";NM$
3000 OPEN "I",#1,NM$
```

Line 3000 sets NM$ as file number 1 and allows the program to receive input ("I") from the file.

If you wanted not only a printout of the sorted checks but also a disk file, you could add the following lines to the program:

```
3100 INPUT "NAME OF FILE FOR SORTED
     CHECKS";SNM$
3200 OPEN "O",#2,SNM$
```

To print the file of selected checks, add the following lines (change PRINT to LPRINT if you want the output sent directly to your printer):

```
3500 INPUT "TYPE OF CHECK TO BE SORTED
     1)MED., 2)TAXES, ETC.:",TNS
4000 INPUT#1,CHKNO,AMNT,NAM$
4020 INPUT#1,MO$,DAY,YR,PURP$,NT
5000 IF TNS< >NT THEN 6000
5100 PRINT CHKNO,AMNT,NAM$
5200 PRINT MO$,DAY,YR
5300 PRINT PURP$
5500 WRITE#2,CHKNO,AMNT,NAM$
5600 WRITE#2,MO$,DAY,YR,PURP$,NT
6000 REM NOT SORTED
6100 IF NOT EOF(1) THEN 4000
7000 REM END OF FILE 1
7500 CLOSE
```

Opening the original file to add new information and make corrections requires more code. I have chosen to write another program for updating the checks file. The new program reads the checks file, allows you to add or correct data, and then writes the output to a new, updated file. (Note that the END statements used below not only terminate the program but also close all files.)

```
1000 REM PROGRAM TO UPDATE FILE
2000 INPUT "OLD FILE NAME";ONM$
2010 INPUT "NEW FILE";NNM$
3000 OPEN "I",#1,ONM$
3010 OPEN "O",#2,NNM$
4000 INPUT#1,CHKNO,AMNT,NAM$
4010 INPUT#1,MO$,DAY,YR,PURP$,NT
5000 PRINT CHKNO,AMNT,NAM$
5010 PRINT MO$,DAY,YR,PURP$,NT
5100 INPUT "UPDATE DESIRED 1)NONE,
     2)CHANGE, 3)ADD NEW CHECK:",CD
5102 ON CD GOTO 5130, 5120, 5110
5104 END
5110 GOSUB 7000
5120 GOSUB 6500
5130 GOSUB 7000
6000 IF NOT EOF(1) THEN 4000
6100 REM END OF FILE DETECTED
6200 END
6500 INPUT "NEW CHECK NO., AMOUNT,
     NAME:",CHKNO,AMNT,NAM$
6510 INPUT "NEW MONTH, DAY, YR, YR,
     PURPOSE, TYPE:",MO$,DAY,YR,
     PURP$,NT
6520 RETURN
7000 WRITE#2,CHKNO,AMNT,NAM$
7010 WRITE#2,MO$,DAY,YR,PURP$,NT
7020 RETURN
```

From Key To Screen

Joyce Pope
Longview PC Users Group

Pressing a key on your computer keyboard usually results in the immediate appearance of a character on the monitor screen. However, a lot happens between your fingers and the screen. When a key is pressed or released, the I/O (input/output) chip in the keyboard determines which key was pressed and for how long. It then passes a signal to the system's I/O chip, which passes it to the CPU (central processing unit). The CPU either passes the signal on to a small program in ROM (read only memory) or passes it to an application program that has redefined the keyboard key definitions. The program receiving the signal runs a routine to determine the appropriate code or codes to send to the screen RAM (random access memory) where the computer stores all codes used to build the current screen image.

Meanwhile, the video circuitry is constantly scanning the screen RAM. By getting the bit patterns for each letter from a character ROM, it sends the appropriate signals to the monitor, which draws the character on the screen in the right place and, if selected, in the right color. All this activity happens in a few millionths of a second.

New Directory Available

Karen Porterfield
IBM Corporation

To counter rising software costs, IBM offers personally developed software through its catalogue *The Directory of Personally Developed Software*. Due to an enthusiastic response to the first two editions, IBM now presents the third edition of *The Directory*.

The expanded *Directory* includes affordable, high-quality, high-function software and has additional innovative program selections.

The Directory features 76 Personal Computer programs, including 22 new products and three Special Holiday Packages. There's software for learning, personal productivity, managing your business, programming, and —for the first time—lifestyle products. All the offerings reflect the value that subscribers have come to expect from *The Directory*.

While programs are priced from $14.95 to $249.95, over half the products cost less than $20. To order *The Directory* free, call 1-800-IBM-PCSW or write to:

Personally Developed
Software
P.O. Box 3280
Wallingford, CT 06494

By completing and returning the survey form enclosed in the catalogue, you will automatically receive the next edition of *The Directory* free.

The three Special Holiday Packages offer an opportunity to save an additional 30 percent on the cost of personally developed software. There's an Education package that offers six programs for $99, a savings of $50.70 off the regular price:

- Algebra Tutor
- Beyond Basic BASIC
- The Combined Adventures With Numbers
- FORTRAN Tutor
- Private Tutor Presenter 1.10
- Word Seeking

An Entertainment package offers six programs for $99, a savings of $45.70:

- Alley Cat™
- Bridge Break 200
- M.U.L.E.™
- Sports Appeal
- Trivia 103
- The World's Greatest Baseball Game™

A Word Processing package features five programs for $99, a savings of $65.75:

- PC Paint
- Personal Editor II
- Personal Print Control
- Select-A-Font
- Word Proof II

To receive the Special Holiday Package discounts, your order must be placed by December 31, 1985.

The Directory has these advantages:

- Direct ordering by phone or mail
- Low-cost, high-quality programs tested by IBM employees.
- Online program documentation to guide users through programs quickly and easily.
- Discounts on quantity orders when you order 50 or more of the same title (excluding Value Packages).

The following new products are available from *The Directory of Personally Developed Software*.

BUSINESS FAMILY

Programmed Evaluation of Contract Option Strategies
A valuable tool for analyzing listed stock and stock index options, this program is a complete toolkit of analytical techniques previously available only to the options-market professional at a premium price. You can use a system-provided set of widely used strategies or allocate space for your own. You can sort and compare the results of previously saved strategies. You can retain the most attractive strategies on the screen. A Relative Performance Index is computed for each of the strategies and is automatically recomputed as the price range changes. You can display information screens showing potential profit or loss, break-even points, return rate indices, commissions, days to expiration, dividends, and margin requirements for each strategy. You can SCAN for the best option values, analyze fair market value and hedge rations via the Black-Scholes valuation model. ($249.95)

RealEstate Investment Package
This program includes 17 spreadsheet templates that provide the information you need to make decisions in today's changing real estate market. Now you can analyze your investments before you sign on the dotted line.

Each template, when combined with the "what if" capability of spreadsheet software, produces accurate answers in seconds.

You don't need to remember complex formulas. Templates are provided for eight-year investment analyses, amortization, depreciation, income/expense tracking, and lease versus purchase analysis. ($19.95)

EDUCATION FAMILY

The Combined Adventures With Numbers
This program is a math adventure game for "young people" who can explore up to 12 castles while solving math problems. ($39.95)

PC Morse Code
This program breaks down the alphabet, numbers, and special characters into "bite-sized" groups that let you learn and practice Morse code easily. ($19.95)

ENTERTAINMENT FAMILY

Trivia 103 (Challenges for Young People)
This game brings Trivia excitement for "kids" from 8 to 80. There are 5,000 questions covering 150 categories. ($19.95)

Sports Appeal (Trivia from Billiards to Baseball)
This game features more than 5,000 questions in over 125 topics from the world of athletics. ($19.95)

Alley Cat™
Trying to steal a kiss from his girl friend Felicia is quite an ordeal for Freddy the Cat. He must first get through seven different mini-games in which, among other feats, he must catch mice, fish, and birds, while avoiding such dangers as a junk yard dog (Bowser Von Spike), an electric eel, a very poisonous spider, and Felicia's protective brothers.

This arcade-style game with delightful color graphics offers four skill levels, keyboard or joystick support, a pause mode, an original music theme (three voice music on the PCjr), and an on/off option for the sound. ($24.95)

M.U.L.E.™
This is a fun-filled strategy game based on free enterprise economics. You use a Multiple Use Labor Element (M.U.L.E.) to help develop your land on the distant planet Iraton. Players are given money and materials to start with, and have six "months" to play. Players can develop their land for farming, energy production or mining. The game has four levels of action. Players can see status reports of their own resources and the resources available at the general store. Each level introduces more complex activities such as land auctions and fluctuating resource prices. The tournament level is extremely fast-moving and requires fast decisions. ($29.95)

The World's Greatest Baseball Game™
The World's Greatest Baseball Game is a baseball simulation game based on real statistics. You can play The World's Greatest Baseball Game three ways: player against player, player against the computer, or participate as the computer plays itself. In each mode, you can choose from a large list of teams, featuring recent World Series and All Star teams, many National and American league teams, and classic teams of the past like Babe Ruth's Yankees. This

allows you to play a team from one era (1961 New York Yankees) versus a team from another era (1984 Detroit Tigers) using actual team statistics.

No game is ever the same, however, because you can observe, participate or manage. In any mode, the two teams play each other, pitching, batting, and fielding the ball. If you are in control, you choose where the pitcher throws the ball. The batter presses a button to hit the ball when it crosses home plate. Fielders are positioned with quick sequences of joystick moves, and runners can steal after the pitcher throws the ball.

The Major League Player's Association has granted permission to use the players' actual names. ($24.95)

LIFESTYLE FAMILY

Bridge Break 200
Play up to 200 hands created by two-time World Bridge Champion Mike Lawrence or choose the Autoplay option to have the computer play as you see on-screen explanations of key plays. ($24.95)

PC Checkbook
This program is designed to help you manage your checkbook. It prints a simple transaction, helps reconcile your checking account with the bank statement, and prints checks, if you use computer check stock paper. ($19.95)

Checkbook I/O
This high-function program helps you set up a budget and maintain up to five different checkbooks and cash books. It even reminds you when your bills are due. ($29.95)

PRODUCTIVITY FAMILY

Personal Editor II
This text editor can be customized into a personalized word processor. Options include variable tab and margin settings, word wrapping, and text formatting. New features include the ability to assign commands to keys, DOS path support, and a split screen that allows you to divide your screen into two or four "windows" so you can work with more than one file at a time or four parts of the same file at once. ($49.95)

Editor's note: See the article "What's New in Personal Editor II" elsewhere in this issue.

PrintDW
This program enables you to convert IBM DisplayWrite print files to standard ASCII files without embedding printer controls, as well as to manipulate ASCII files. This allows you to take advantage of function-rich DisplayWrite word processing programs to produce documents for use with compatible programs, printers, and host computer software. ($19.95)

DOS Memories
This program offers full-screen console capabilities to DOS, allowing you to scroll and edit commands on previously displayed screens. Any screen displayed by either a DOS command (e.g., TYPE or DIR) or a compiler can be reviewed and edited. DOS Memories remembers your previously entered commands and lets you recall them with one keystroke. You can edit these commands and assign often-used commands to keys. ($34.95)

Multimedia Presentation Aid
Make your presentations come alive with color. You can use this program with a text editor to create presentations for viewing on the PC. You can scroll presentations forward and backward or set up a "timed" operation. You can prepare multicolor transparencies using a plotter connected to an IBM PC through an Asynchronous Communications Adapter. Using an IBM printer, you can produce transparency-quality presentations. ($24.95)

Personal Correspondence Manager
This program gives your personal computer greater flexibility and versatility to create, view, proof, file, and print documents. Repetitive work is reduced when you can save often-used names and addresses to be used in different letters, memos, and other correspondence. IBM SCRIPT/PC, IBM Personal Editor, or IBM Word Proof can be started with a single keystroke. ($39.95)

PC Print
With this program, you can print reports or other documents using a variety of font selections and options. You can create and edit fonts and produce output sideways as well as normally. PC Print's powerful commands can be run from the program menu, DOS prompts, or BATch files. You can print ASCII files from editors, BASIC, or assembler language source code documents. Print command options are easily built from menu selections and quickly started with a function key. ($24.95)

JoyMouse
With JoyMouse, you can substitute a joystick for your keyboard to move the cursor with the speed of a mouse. JoyMouse allows a joystick connected to your IBM Game Control Adapter to be used for cursor movements. You can even define the joystick buttons to perform the tasks of up to three function keys or key combinations. You can assign one of nine different speeds for cursor movement and redefine joystick buttons for different programs ($14.95)

PROGRAMMING FAMILY

Language Extension System Assembler
Your Macro Assembler programs can access the 8087 or 80287 math processors to do equations faster and more accurately. You can use macros to include 8087/80287 opcodes in your Macro Assembler output. ($24.95)

PCWATCH
Now you can zero in quickly on programming trouble spots. This powerful debugging tool lets you monitor computer activity at the system interface level. You can trace external interrupts, BIOS system functions, DOS services, and user-defined functions built around the INT instruction. ($39.95 until December 31, 1985, $49.95 after)

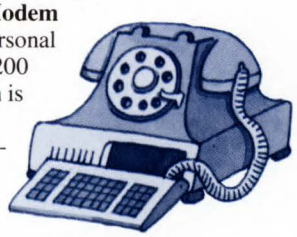
Structured BASIC Facility
Turn BASIC into a structured language to help you work faster. The Editor helps you create and edit programs with line and full-screen editing capabilities. The Preprocessor converts Structured BASIC into source files for both the BASIC interpreter and Compiler. The Generator converts existing BASIC programs into structured BASIC source files. ($54.95)

Additional New Products

Following is a listing of new products that IBM has recently announced.

Hardware

IBM Personal Computer 1200 BPS Modem



The IBM Personal Computer 1200 BPS Modem is a half-card sized, microprocessor based modem designed for Asynchronous data transmission at speeds up to 1200 BPS. The modem supports point-to-point configurations in duplex mode and is compatible with Bell 212A (asynchronous only) and 103 practices, and with CCITT V.22B (asynchronous only) recommendation for transmitting over the public switched telephone network.

Automatic dialing is supported along with most of the "intelligent" Attention (AT) commands popular within the industry. The modem includes a completely unique set of commands which permit various diagnostics to be executed under the control of the IBM Personal Computer. These commands also control speed selection, V.22 operational mode, call progress reporting and help functions. Both command sets are built-in functions of the modem.

In addition, the modem supports:

- Asynchronous data transmission in full duplex mode at speeds up to 1200 BPS
- Automatic and adaptive equalization performed by V.22 configurations of the modem and continuing to adapt while in data mode
- Automatic generation of 2100 hertz answer tone in V.22 mode or 2225 hertz in Bell 212 mode
- Automatic or manual answer capability
- Control for A/A1 telephone leads in applications with Key Telephone Systems
- Automatic speed detection for DTE transmission speeds of 75, 100, 110, 134.5, 150, 200, 300, 600 or 1200 BPS and adjusts for compatibility
- Automatic or manual dial
- Tone or pulse dialing
- Automatic redial

- Audio monitoring of the call in progress via the IBM PC speaker with command driven volume control
- Automatic detection of tones for dial, busy, ringback and attention
- Automatic detection of voice or failed call
- Data quality indication can be accessed when receiving data in Bell 212 or V.22 modes

The IBM Personal Computer 1200 BPS Modem can be installed in the IBM Personal Computer, IBM Personal Computer XT, IBM Portable Personal Computer, or IBM Personal Computer AT. It requires one feature slot and comes with a 15 foot cable to connect the modem to a modular phone jack. A customer-supplied handset can be attached.

IBM Quietwriter Printer model 2
The IBM Quietwriter Printer model 2 has all the functions of the original Quietwriter plus high-resolution, all-points-addressable graphics compatible with the IBM Personal Computer. The IBM Quietwriter printer features quiet, non-impact letter quality printing in four pitches, and multiple type styles. The Quietwriter Printer holds two removable font cartridges, each supporting the full IBM Personal Computer character set. A cut-sheet feeder or continuous forms feeder is optional.

The Quietwriter Printer prints at up to 40 to 60 characters per second burst speed, and has a 13.2 inch writing line. Graphics are compatible with the IBM Graphics Printer and IBM Proprinter. Graphics resolutions supported are 72 x 60, 72 x 120, 72 x 240, 60 x 60, 120 x 120 and 240 x 240. The Quietwriter Printer can be directly attached to any member of the IBM Personal Computer Family with the appropriate printer adapter. Owners of the Quietwriter Printer model 1 may purchase an upgrade to a model 2 through their IBM Service Repair Center or Authorized IBM Dealer.

In addition, IBM announced the IBM Quiet Non-Correcting Ribbon. This ribbon cannot be erased without detection, and is therefore suitable for checks and other negotiable documents.

Software

Biology Series 11-20
The Biology Series is a set of integrated science programs designed to help high school students learn the major concepts and processes found in modern biology. Highly varied and interactive instructions ensure high student involvement. An investigative approach coupled with thinking skills and an emphasis on scientific processes make these programs valuable tools in biology classrooms and laboratories. Each program contains several lessons and one testing section. Other features include:

- Animation to illustrate major concepts and processes
- Simulation to show experimentation
- Interactive sequences for student participation
- Problems to encourage reasoning skills
- "Learning loops" for reinforcement of difficult concepts
- Quizzes and tests to promote subject mastery
- Instructional guides

Each program comes in a Single Pack which contains one program diskette and one instructor's manual. Also available are School Packs which contain 12 program diskettes and one instructor's manual.

The Biology Series programs require an IBM Personal Computer, IBM Personal Computer XT, IBM Portable Personal Computer, IBM Personal Computer AT or IBM PCjr with 128KB of memory; one double-sided diskette drive; the IBM Color Graphics Display, IBM Enhanced Color Display or IBM PCjr Color Display with the appropriate adapter; and DOS 2.00 or higher (DOS 3.00 or higher on the Personal Computer AT).

Chemicals of Life III:
Proteins and Nucleic Acids
This program describes how proteins and nucleic acids are formed and how these biochemicals function in living organisms. This course teaches students dehydration synthesis reactions, hydrolysis reactions, the structural formula for amino acids, protein synthesis, proteins and their functions in living organisms, enzymes and coenzymes and dipeptide bonds.

Mendelian Genetics:
The Science of Inheritance
The laws of heredity governing gene inheritance are covered in this program. Laws including independent assortment, segregation, dominance, codominance, recession and sex-linkage are covered. Students learn about dominant and recessive traits, crossing genes that show codominance, phenotypic ratios for hybrid crossing, laws of segregation with chromosomal models in meiosis, gene linkage and gene studies of various organisms.

Regulation and Homeostasis:
Systems in Balance
This program describes the natural processes that regulate the internal and external environments of living systems. The program explores body temperature versus environmental temperature and body regulation of oxidation. It discusses the role of the brain in the regulatory processes of homeostasis and feedback as well as which organs in the human body regulate what conditions, and what actions the nervous and endocrine systems take in regulation and the function of hormones.

Cytology and Histology:
Cells and Tissues
This program explains the structure and function of cell organelles and how cells work together to form tissues. Students learn the similarities and differences between animal and plant cells in structure and function. Lessons identify the difference between living and nonliving

things, and the relationship between cell structure and specialized function in an organism. Students learn to identify the organelles of plant and animal cells, the function of those organelles and the function of parts of unicellular organisms.

Human Life Processes II: Systems Level
This program describes complex tissue and organ functions, such as respiration, digestion and excretion. Students study the cell as a basic structural unit of the body, specialization of cells in multicellular organisms as tissues, major tissues of the human body, organs like the stomach and their tissues, and the role of the major organs and systems of the body.

Students learn about internal and external respiration and how cells obtain energy. Students trace the passage of oxygen and waste molecules through the respiratory system, identifying the parts. The parts of the kidneys and urinary system are labeled as students define the filtration and selective reabsorption that occurs in the kidneys.

The role of the nervous system is explained. Students see examples of reflexes and learn the various sense organs of the body and their stimuli and response. Students cover the parts of the neuron as well as the major parts of the nervous system and their functions including the structure of the spinal cord.

The Environment I: Habitats and Ecosystems
The Environment I: Habitats and Ecosystems shows students how abiotic factors influence biotic factors in populations and communities within ecosystems. Students learn about ecology and the effects that light, temperature, moisture, soil and air have on community organisms. Biotic factors like predation, competition and symbiosis are explained in light of their effect on organisms and species dominance. Students learn the terms community, niche, biosphere, population and biotic potential. Students explore numerous terrestrial and aquatic biomes.

Physics Discovery Series 1-4
The Physics Discovery Series 1-4 is a set of programs that helps physics students understand and investigate the physical processes at work in their surroundings. Each program uses highly interactive instruction to achieve high student interest and comprehension. Emphasizing scientific investigation and thinking skills, these courses provide valuable tools for physics classrooms and laboratories.

Each program has three modules. The first lets students change variables to discover physical relationships. In the second, students analyze phenomena, develop formulas, and record and graph data. In the third module, students apply physical laws and concepts to problems and investigations. For advanced students, an additional section offers more challenging activities.

Each program is available in a Single Pack version which includes one program diskette and one guidebook, or as a School Pack with 12 copies of the program diskette and one guidebook.

The Physics Discovery Series 1-4 requires an IBM Personal Computer, IBM Personal Computer XT, IBM Portable Personal Computer, IBM Personal Computer AT or IBM PCjr with at least 128KB of memory; an IBM Color Display or IBM PCjr Color Display and appropriate adpater; one double-sided diskette drive and DOS 2.00 or higher (DOS 2.10 for the PCjr and DOS 3.00 for the Personal Computer AT). A printer is optional.

Investigating Acceleration
Investigating Acceleration teaches the laws and concepts of acceleration. Students cover motion in terms of distance, velocity and acceleration and design experiments to measure the variables that affect acceleration. The students then determine the formula that expresses the relationship between acceleration and its related variables. Students learn to calculate force, mass or acceleration, given the appropriate information and apply Newton's Second Law of Motion to commonly observed phenomena.

Investigating Gravitational Force
This program lets students explore gravitational force and determine the variables that affect it. Students design experiments to measure these variables, analyze graphs and determine the mathematical formula that expresses the relationship between gravity and the variables that affect it. Students apply the Universal Law of Gravity to forces on masses on the earth's surface and develop hypotheses about the effects of different gravitational forces on the earth.

Investigating Conservation of Energy
Investigating Conservation of Energy lets students explore, analyze and apply the laws and concepts of energy conservation. They determine which measurable variables affect the potential and kinetic energy transfer, and they design experiments, analyze graphs and draw conclusions based on the graphs. Students determine the mathematical formula that expresses the relationship between mass, height, and acceleration in gravity to the potential energy of an object. Students calculate the potential and kinetic energies of an object based on given variables. They also apply the potential and kinetic energy relationships as well as applying conservation-of-energy principles to commonly observed phenomena.

Investigating Thermal Energy
Students study the concepts and laws of thermal energy, determining the variables which affect thermal energy transfer. Students design experiments to determine the specific heat of common materials. They determine the formula that expresses the relationship between heat, mass, specific heat and change in temperature, then perform calculations based on appropriate information. Experiments and observations of common phenomena are included.

Exchange Mirrors User Group Growth

Expanded Support
You're growing and so is IBM PC User Group Support! From 1983 to 1984, the number of user groups registered with us nearly tripled. From 1984 to 1985, that number nearly doubled. These growing numbers have been paralleled by the growing support and services User Group Support offers to registered user groups.

Consider this publication. We are now publishing this expanded version of *Exchange* in a magazine format. *Exchange*'s attractive design and layout presents to you the information we print as invitingly as possible. Articles are carefully chosen to give you information from IBM as well as to pass along valuable information from user groups around the country. And each month our distribution numbers increase.

More Departments
In *Exchange* we include a Departments section, with various departments such as New Products, Ask IBM, and Editor's Comments. In the future, we'd like to add a Letters column featuring letters from you. We welcome any suggestions or comments you may have about *Exchange,* IBM software and hardware products, or other related subjects. Your feedback will help *Exchange* remain a healthy publication and allow us to stay attuned to your needs. You can write to us at:

Editor, *Exchange*
IBM Corporation (2900)
P.O. Box 3022
Boca Raton, FL 33431-0922

Your Input Counts
We're delighted with the favorable responses to *Exchange* that we have seen so far in your user group newsletters. We're glad you like *Exchange*!

It takes both of us to make *Exchange* a success—and together we make a great team. So, keep writing articles for your own user group newsletters and let us hear your comments or suggestions about this publication. You can make *Exchange* even better and ensure that it truly serves your needs.

Karen Porterfield
Associate Editor

Copyrights, Trademarks, and Service Marks

"In conclusion, I liked the machine. I thought it was wonderful to be able to run a local session of CMS and share minidisks between the host and local machines. (page 2)

"In PE2, you may assign definitions to 238 typeable keys, plus 76 user-shifted keys. (page 4)

"PE2 let you split the screen of its display into as many as four windows. Each window contains a section of any file. (page 5)

"The algorithm for screen updates was substantially revised. PE2 defers all changes to the screen until all commands in a macro or a key definition have finished executing. (page 7)

"The Indexed Sequential Access Method (ISAM) is a library of subroutines that lets you access files sequentially or randomly by an *index*. (page 11)

"You should describe each field in the record when you create an ISAM file. This provides an easy way to identify each file and its contents. (page 12)

"Because BASIC Compiler version 2.00 allows you to compile and link separate subprograms, you have a flexible environment for structuring large programs. (page 15)

G320-0846-00