Volume 3, Number 2 June 1985

IBM Personal Computer Seminar Proceedings

**The Publication for Independent Developers
of Products
for IBM Personal Computers**

**Published by International Business Machines Corporation
Entry Systems Division**

IBM®

Changes are made periodically to the information herein; any such changes will be reported in subsequent Proceedings.

It is possible that this material may contain reference to, or information about IBM products (machines and programs), 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 products, programming or services in your country.

IBM believes the statements contained herein are accurate as of the date of publication of this document. However, IBM makes no warranty of any kind with respect to the accuracy or adequacy of the contents hereof.

This publication could contain technical inaccuracies or typographical errors. Also, illustrations contained herein may show prototype equipment. Your system configuration may differ slightly. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

All specifications are subject to change without notice.

Printed in the United States of America

Contents

Introduction and Welcome

These are the Proceedings of the IBM Personal Computer Seminar, designed for independent developers of products for IBM Personal Computers. The purpose of these Proceedings is to aid you in your development efforts by providing relevant information about new product announcements and enhancements to existing products. This issue is prepared in conjunction with this seminar. The Proceedings of future seminars for the IBM Personal Computers also will be published and will cover topics presented at those seminars.

Throughout these Proceedings, the term Personal Computer and the term family of IBM Personal Computers address the IBM Personal Computer, the IBM Personal Computer XT, the IBM PC*jr*, the IBM Portable Personal Computer, and the IBM Personal Computer AT.

Purpose

What is our purpose in issuing a publication such as this? It is quite simple.

The IBM Personal Computer family is a resounding success. We've had a lot of help in achieving this success, and much of it came from the independent developers.

As you proceed with your development, do you at times wish for some bit of information or direction which would make the job easier? Information which IBM can provide? This is the type of information we want to make available to you.

Since we want to be assured of giving you the information you need, we ask you to complete the questionnaire which appears at the end of these Proceedings. Your response to this questionnaire will be taken into account in preparing the content of future issues, as well as the content of seminars we will present at microcomputer industry trade shows.

Topics

The following list gives a general indication of the topics we plan to cover in future seminars and include in the IBM Personal Computer Seminar Proceedings:

- Information exchange forum — letters to the editor format
- Development tools — languages, database offerings
- Compatibility issues
- New devices — capacities and speeds
- System capacities — disk and memory
- Enhancements in maintenance releases
- Tips and techniques
- New system software
- Hardware design parameters
- Tips on organizing and writing documents for clear and easy reading
- Changes to terms and conditions

The BASIC Compiler 2.00

The BASIC Compiler 2.00 is an optimizing compiler designed to complement the BASIC Interpreter. (Optimizing compilers do such things as change the order of expressions or eliminate common sub-expressions to either improve performance or decrease the size of the programs.)

Creating application programs with the IBM Personal Computer BASIC Compiler 2.00 provides several benefits, some of which are:

- Networking support (Lock and Unlock statements)
- Full graphics support
- Full PC*jr* compatibility
- Increased file capacity (16,775,616 record maximum- formerly 32,767)
- Increased input string length (32,767 character maximum- formerly 255 character maximum)
- Full DOS file capability
- Shell commands support
- Line numbers unnecessary
- Separately compiled modules allow creation of larger programs
- Increased execution speed for most programs when compared to the interpreter version
- BASIC source code security

The BASIC Compiler 2.00 offers a powerful programming environment in which you can use the BASIC Interpreter to quickly run and debug programs and then later compile those programs to increase their execution speed.

A compiled program is optimized machine code, not source code. Consequently, compiling substantially improves execution time and protects your source program from unauthorized alteration or disclosure.

Licensing Agreement

Application programs that require the runtime modules BASRUN20.EXE, REBUILD.EXE, or ISAM.EXE, *cannot* be distributed without entering into a license agreement with IBM. A copy of the license agreement can be obtained by writing to:

IBM Corporation
P.O. Box 2910
Delray Beach, Florida 33444
Attn: Personal Computer Customer Relations

Note, however, that by compiling with the **/O** parameter, it is possible to develop programs with the BASIC Compiler 2.00 that do not use the BASRUN20.EXE runtime module and, therefore, do not require the license agreement. This does not apply to ISAM.EXE or REBUILD.EXE.

Hardware Requirements

The hardware necessary to use this product is:

- Any of the following IBM Personal Computers:
  - IBM Personal Computer
  - IBM Personal Computer XT
  - IBM Personal Computer AT
  - IBM *Portable* Personal Computer
  - IBM PC*jr*.
- A minimum of 128K bytes of Random Access Memory (RAM)

  Note—Additional memory can significantly improve the performance of the BASIC Compiler 2.00 and the Linker when used on all of the above listed computers
- One or two double-sided diskette drives or a fixed disk
- A printer (highly recommended)
- A display screen

  Although various displays can be used, best results are obtained with an 80-column display.
- Blank, formatted diskettes

Software Requirements

The software necessary to use this product is:

- Disk Operating System (DOS) 2.1 or later version

Changes in BASIC Compiler 2.00

BASIC Compiler 2.00 differs from BASIC Compiler 1.00 in the following areas:

- Improved program control structures allow a more modular approach to programming. Enhancements include:
  - Named subprograms - provides the ability to call (execute) a routine or subprogram by a name instead of a line number.
  - Named COMMON blocks - can be used for intermodule communication without chaining. Items listed in blank COMMON can be accessed by another chain file.
  - User-defined multiline functions - permits a function definition to occupy more than one program line. It must begin with a DEF FN statement and end with an END DEF statement.
  - Ability to branch to alphanumeric labels - it is no longer necessary to use only line numbers; now meaningful statement labels may be used (example: GOTO TOTALS).
  - Separately compiled BASIC subprograms.
- Larger programs can be compiled. The use of a memory model that separates the instruction space from the data space allows this, as well as allowing more than twice as much symbol table space. Please note, however, that the data segment has a maximum upper limit of 64K bytes. In addition, allocated string space is also limited to a maximum of 64K bytes.
- Large dynamic arrays are supported. The maximum index for any dimension of a numeric array is 32767. This dimension limit and the amount of memory in your machine are the only size restrictions for numeric arrays.
- .EXE files produced by BASIC Compiler 2.00 are larger than those produced by BASIC Compiler Version 1.00.
- Graphics capabilities are expanded. All graphics features of the BASIC Interpreter are available. These include the following statements:
  - VIEW
  - WINDOW
  - PMAP
  - LINE
  - DRAW
  - POINT
  - PAINT
- Access to DOS is expanded. Several new features of the BASIC Interpreter are available to allow more flexible use of DOS functions. Statements affected are:
  - SHELL
  - IOCTL
  - IOCTL$
  - ENVIRON
  - ENVIRON$
  - ERDEV
  - ERDEV$
  - MKDIR
  - RMDIR
  - CHDIR
- The *filespec* syntax is expanded to allow the specification of a path for a device or file.
- Redirection of standard input and standard output is supported.
- Enhanced event trapping is available. This enhancement affects the following statements:
  - ON TIMER
  - ON PLAY
  - ON KEY

- All advanced features of PC*jr* BASIC are supported. The full range of sound and graphics capabilities are available to users of PC*jr*. Some of the features include:
  - PLAY – Multi-voice
  - PLAY – Volume Control
  - NOISE
  - Enhanced SCREEN statement
  - Enhanced CLEAR statement
  - PCOPY
  - User-defined PALETTE
  - Additional screen modes
- Compiler termination codes are returned when the compiler exits. These termination codes can be tested by the IF batch subcommand of DOS.
- An input editor is included. Input required by your program can be altered easily on the screen.
- Support is provided for up to five levels of nested $INCLUDE files.
- When compiling, you must specify the **/D** parameter to Ctrl-Break effectively at runtime.
- BASIC library files are searched in the following order:
  1. User-specified directory
  2. Current directory
  3. PATH directories
  4. User-prompted directory.
- Graphics statements now use line clipping instead of wraparound.
- The OPEN statement has been enhanced to include file access control.
- Because of the added functions in BASIC Compiler 2.00, you may notice slightly longer compile and link times.

BASIC Compiler Version 2.00 includes the following language additions:

Statements

CALLS — Calls and transfers program control to IBM Personal Computer Macro Assembler routines.

DEF FN, END DEF, EXIT DEF — Designate the beginning and ending of a multiline function.

LOCK, UNLOCK — Restrict access by other processes to all or part of an opened file.

REDIM — Changes the space allocated to a dynamic array.

SHARED — Designates variables as global to the subprogram and the calling program.

STATIC — Designates variables as local to a subprogram or multiline function.

SUB, END SUB, EXIT SUB — Designate the beginning and ending of a subprogram.

Functions

COMMAND$ — Returns the parameters from the command line used to invoke the current program.

LBOUND — Returns the value of the lowest subscript available (either 0 or 1) for any array. This value depends on the setting of the OPTION BASE statement.

UBOUND — Returns the value of the largest subscript available for any array.

New File Type — ISAM

The BASIC Compiler 2.00 now supports the indexed sequential access method. These ISAM files are accessed through the CALL statement. ISAM files allow for rapid access to large files by key values. Other features are automatic storage space management and fast sequential access.

Library Manager

The IBM Library Manager is included. This utility enables you to construct and edit object module libraries. See Appendix E, of the *BASIC Compiler Fundamentals Manual* for details.

Differences Between the Compiler and Interpreter

Differences between the languages supported by the BASIC Compiler 2.00 and the BASIC Interpreter must be taken into account when compiling existing or new BASIC programs.

The differences between the languages supported by the BASIC Compiler 2.00 and the BASIC Interpreter are described below:

Operational Differences

Some BASIC commands and statements used to operate in the interpreter programming environment are not acceptable input to the compiler. These are:

AUTO	LOAD
CONT	MERGE
DELETE	NEW
EDIT	RENUM
LIST	SAVE
LLIST	

Certain statements function similarly in the BASIC Compiler 2.00 and the interpreter, but require special parameters to be specified when used with the compiler.

- **Event trapping**: If you use any of the event trapping statements, you must specify either the **/V** or the **/W** parameter when you start the compiler. The event trapping statements are:

COM(n)	ON STRIG(n)
KEY(n)	ON TIMER
ON COM(n)	PEN STOP
ON PEN	PLAY(n)
ON PLAY	STRIG(n)

- **Error trapping**: If you use an ON ERROR statement and some form of a RESUME statement, you must specify either the **/E** or the **/X** parameter when you start the compiler. If you use only the RESUME *line* form, you should specify **/E**. If you use RESUME NEXT, RESUME 0, RESUME, or any combination of those with RESUME *line*, the **/X** parameter must be used instead.

- **Debug code (TRON and TROFF)**: To use TRON and TROFF, the **/D** parameter must be specified when you run the compiler. Otherwise, TRON and TROFF are ignored and a warning is generated.

Note that using these parameters increases the size of the .OBJ, and .EXE files. See the *BASIC Compiler Fundamentals* for a detailed explanation of each of the compiler parameters.

Language Differences

If your machine has a cassette port, the BASIC Compiler 2.00 supports cassette I/O. However, to enable cassette I/O, you must specify the **/O** parameter at compile time and then link the IBMCAS.OBJ module.

Some differences exist among the commands, statements and functions of the BASIC Compiler 2.00 and BASIC Interpreter. These differences are explained in the *BASIC Compiler Language Reference* manual.

Other Differences

Other differences between the BASIC Interpreter and the BASIC Compiler 2.00 include the following:

Double-Precision Arithmetic Functions

If you use double-precision operands for any of the arithmetic functions, including the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, and SQR), the BASIC Compiler 2.00 returns double-precision results. In the interpreter, double-precision results are returned if the interpreter is invoked with the **/D** parameter.

Double-Precision Loop Control Variables

The compiler, unlike the interpreter, allows the use of double-precision loop control variables. This allows you to increase the precision of increment in loops.

Expression Evaluation

Mathematical computations have been modified in the compiler for improved speed and accuracy, so there may be slight differences in the results of single-precision or double-precision operations compared to the interpreter.

Also, the BASIC Compiler 2.00 performs optimization, if possible, when evaluating expressions.

During expression evaluation, the BASIC Compiler 2.00 converts operands of different types to the type of the more precise operand.

```
QR=J%+A!+Q#
```

The above expression causes J% to be converted to single-precision and added to A!.

This double-precision result is added to Q#.

The BASIC Compiler 2.00 is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 40000 for M.

```
I%=20000
J%=20000
M=I%+J%
```

That is, J% is added to I%. Because the number exceeds the 32767 limit for integers, the interpreter converts the result into a floating-point number. The result, 40000, is found and saved as the single-precision number M.

The BASIC Compiler 2.00, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow may occur. If the **/D** debug parameter is set, the error is detected. Otherwise, an incorrect answer is produced. One possible way to avoid this problem is to use single-precision numbers instead of integers.

Besides the previous type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I%=20000
J%=-18000
K%=20000
M%=I%+J%+K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs.

However, if the compiler performs I%+K% first and then adds J%, overflow occurs. The compiler follows the rules of operator precedence, and parentheses may be used to direct the order of evaluation. *No other guarantee of evaluation order can be made.*

Input Statements

The compiler limits the number of variables read by an INPUT or INPUT # statement to 60.

If you try to enter more than 32767 characters in response to any INPUT or LINE INPUT statement, the compiler makes the computer sound a beep.

Integer Variables

The BASIC Compiler 2.00 can make optimum use of integer variables as loop control variables. To help the compiler produce faster and more compact object code, you should use integer variables as much as possible. For example, the following program executes much faster by replacing I, the loop control variable, with I%, or by declaring I an integer variable with DEFINT.

```
100 FOR I=1 TO 10
110 A(I)=0
120 NEXT I
```

It is also advantageous to use integer variables to compute array subscripts because the generated code is faster and more compact.

Input Editor

When you respond to an input statement in a compiled program, you do not have all the facilities of the BASIC program editor to use. The BASIC Compiler 2.00 does not allow you to change lines anywhere on the screen; you may edit only the current line.

The input editor supplied with BASIC Compiler 2.00 uses a special set of commands to manipulate the text on the screen. These commands are different from the commands used by the editor in the BASIC Interpreter.

Input Editor Commands: All of the editor commands, except Delete, require you to press the control key (Ctrl) in combination with another key.

Ctrl-B	moves cursor back one word.
Ctrl-C	exits program.
Ctrl-E	erases to the end of the current line.
Ctrl-F	moves cursor forward one word.
Ctrl-H	deletes the character to the left of the cursor.
Ctrl-I	inserts spaces from the cursor position up to the next tab position (tabs are set every eight spaces). If the editor is in replace mode, any existing characters will be overwritten.
Ctrl-K	moves the cursor to the beginning of the line.
Ctrl-M	issues a carriage return and enters the line.
Ctrl-N	moves the cursor to the end of the line.
Ctrl-R	toggles the editor from insert to replace mode.
Ctrl-T	toggles the function key display line on and off.
Ctrl-U	erases the entire line.
Ctrl-]	moves the cursor one position to the left.
Ctrl-\	moves the cursor one position to the right.
Del	deletes the character at the cursor.

The following special program editor keys are not supported by the compiler:

Home
Ctrl-Home
Cursor Up
Cursor Down
Next Word (Ctrl-Cursor Right)
Previous Word (Ctrl-Cursor Left)
Ctrl-Break.

If you try to use any of these keys (with the exception of Ctrl-Break) in response to an input statement, the computer will sound a two-tone beep.

Pressing Ctrl-Break in response to an input statement returns you to DOS.

All files are closed and the following message is displayed:

```
STOP in Line xxx of Module
   Modulename at Address ---:---
```

Hit any key to return to system

When a key is pressed, the DOS screen mode is restored.

Number of Files

The maximum number of files that can be open simultaneously is 15. The default value is 3. To increase the number of files to be opened simultaneously you must have the following in your CONFIG.SYS file:

FILES=*xxx*

Where:

xxx is the number of files you plan to have open simultaneously, plus 5, which are used by DOS. The maximum value for *xxx* is 20.

Line Length

The interpreter cannot accept lines greater than 254 characters in length. In contrast to the interpreter, the BASIC Compiler 2.00 accepts *physical* lines of up to 32766 characters in length. (A *physical* line for the compiler is one that ends in a carriage return-line feed.) However, you can make the compiler accept much longer *logical* lines of input by ending the physical lines with an underscore character (underscores in quoted strings or remarks do not count). The underscore tells the compiler to ignore the following carriage return, so all it sees in the carriage return-line feed sequence at the end of the line is the line feed character. The line feed is the line continuation character understood by the compiler. For example, the following two physical lines:

```
100 INPUT "Values for array A"; A(1),_
A(2), A(3), A(4), A(5), A(6), A(7)
```

are read by the compiler as a single INPUT statement, that enters seven values into array A.

It is impractical to use this technique with the program editor in the BASIC Interpreter because each line created with the BASIC Interpreter editor must begin with a number. In addition, BASIC Compiler 2.00 source programs that use this technique cannot be debugged using the interpreter.

PEEKs and POKEs

PEEKs and POKEs into the interpreter work area (such as DEF SEG: POKE 106,0) are interpreter dependent and do not work for compiled BASIC.

Note—PEEK and POKE for dynamic array elements work differently than PEEK and POKE for static array elements. See the PEEK and POKE statements in BASIC *Compiler 2.00 Language Reference* for details.

String Length

Strings can be up to 32767 characters long rather than 255 characters long. Therefore, any string function parameters that identify the location in a string or its length (which can have a maximum value of 255 in the interpreter) can now range to 32767.

The internal storage format for the string descriptor requires four bytes rather than three bytes (low byte, high byte of the length, followed by low byte, high byte of the address). If you use machine language subroutines with string arguments, you have to recode the subroutine to account for this change.

String Space Implementation

The implementation of the string space for the compiler differs from its implementation for the interpreter. Using PEEK, POKE, VARPTR or assembly language routines to change string descriptors may result in a **String Space Corrupt** error.

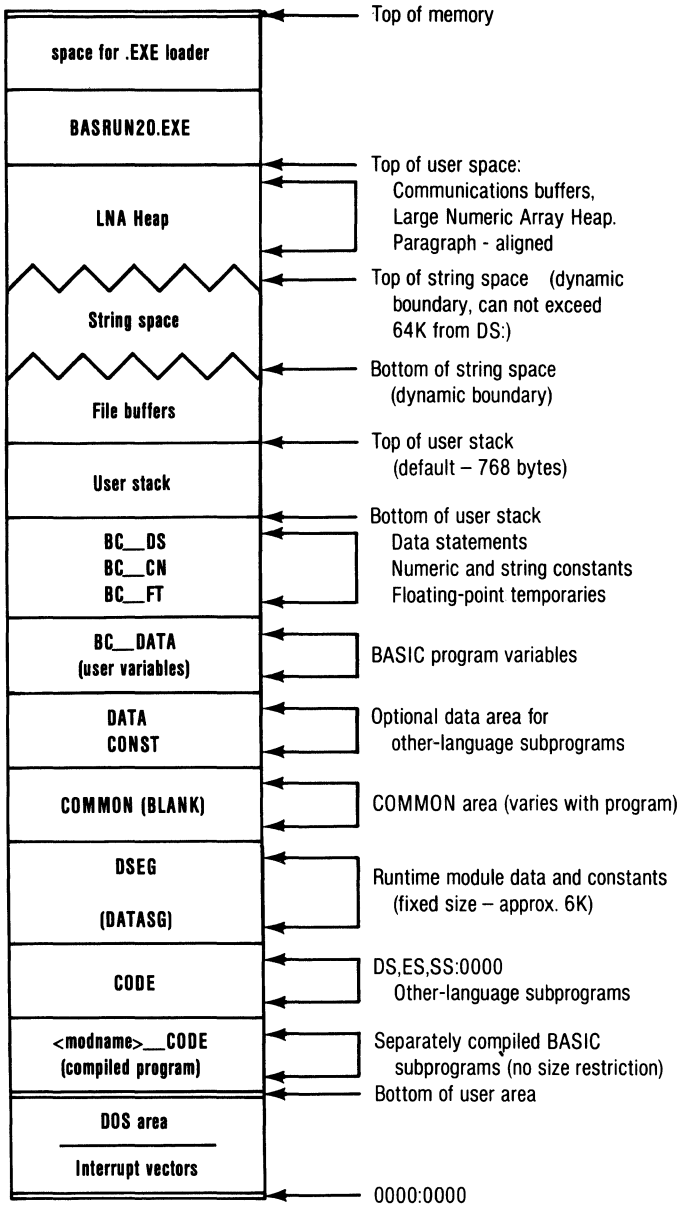
The amount of available string space is 64K bytes. Because of the string descriptor size and the number of internal variables used by the compiler, the number of elements in a string array is less than the amount available in BASIC Compiler Version 1.00 or the BASIC Interpreter. To help maximize the use of memory space, you can move numeric data into dynamic arrays that have a separate data area. Declare string arrays as dynamic and ERASE (reuse) the same space whenever possible. You also can use MID$ to help prevent fragmentation of string space. See the example under MID$ Function and Statement in the BASIC *Compiler 2.00 Language Reference* for details.

Memory Information

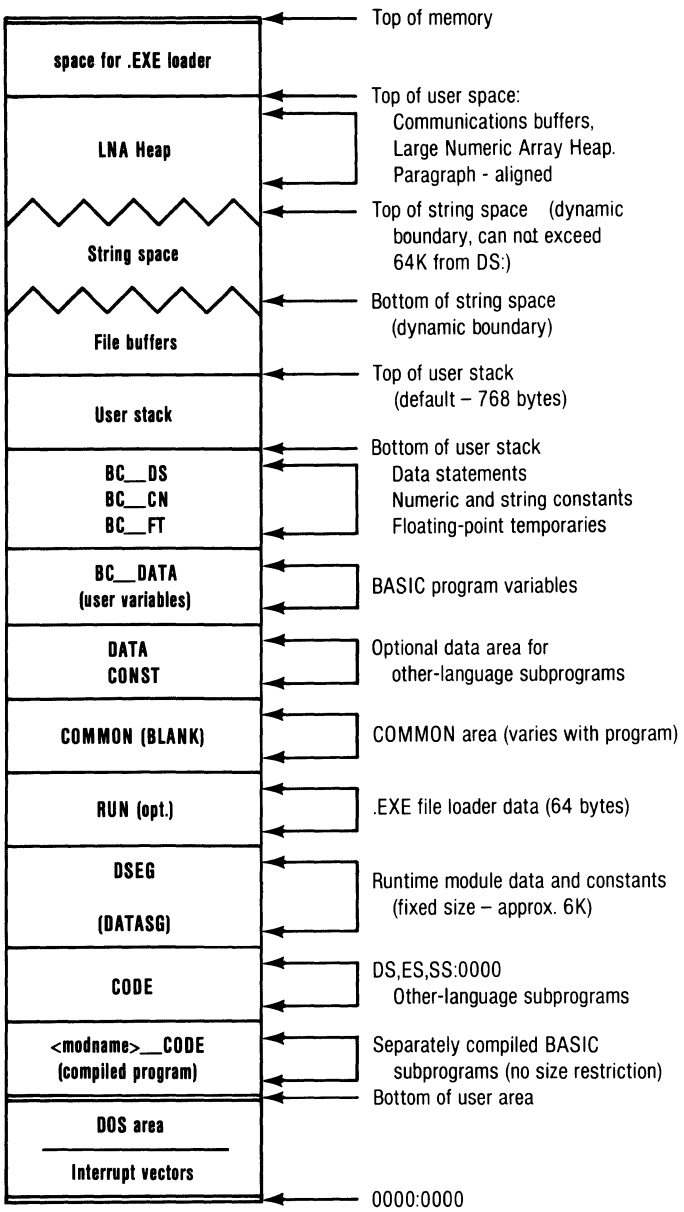
Memory Map

This section contains illustrations of runtime memory maps for programs linked to the two runtime libraries: BASRUN20.LIB and BASCOM20.LIB.

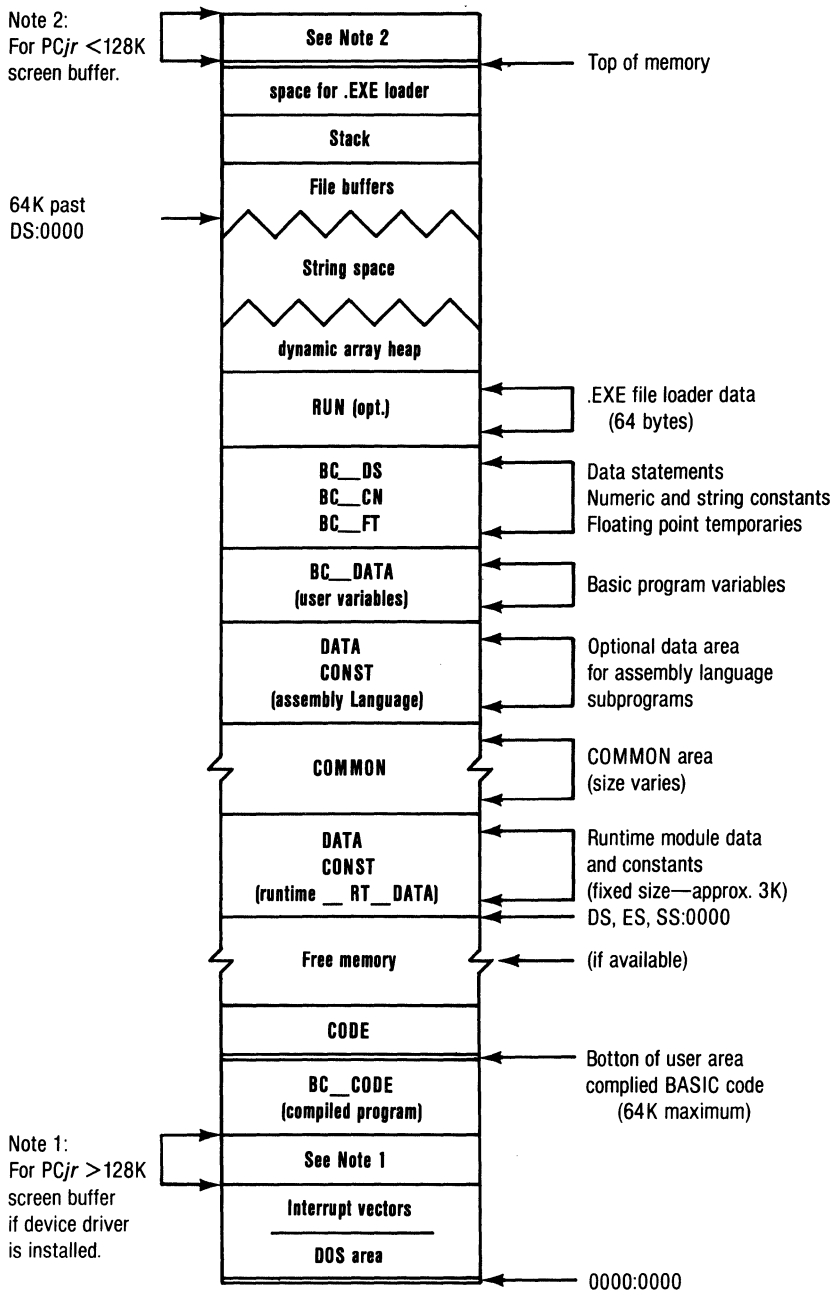
Refer to the IBM Personal Computer *Technical Reference* manual for information that is not in this publication or in *BASIC Compiler 2.00 Language Reference* manual.



Memory Map continued...



PC*jr* Memory Map continued...



Modular Programming Techniques

The ability to compile modules independently and then link them together allows the development of larger, more easily maintainable programs.

Named COMMON Blocks

The BASIC Compiler 2.00 supports named COMMON blocks. 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 respect. Items (variables and/or arrays) in named COMMON blocks are not accessible to separate files that are chained through the use of the CHAIN statement to the module in which they are named.

Items that are declared in the named COMMON blocks of independent modules are accessible by each module if the modules communicate through the CALL statement. Therefore, named COMMON blocks can be used for intermodule communication without chaining. Items that are listed in the blank COMMON statement with no specified *blockname* can be accessed by another chained file.

See the CHAIN and COMMON statements in *BASIC Compiler Language Reference* for details on statement syntax.

If the same blockname is used in more than one module, the items in the item lists must be the same type and size and listed in the same order. However, the names of the variables may be different. For example, the lists A,B,C(2) and E,F,G(2) would be valid, but E,F(2),G would not be valid.

Structuring Modular Programs

The separate compilation capability of the BASIC Compiler 2.00 allows a flexible environment for structuring large programs through the subprogram, module and named COMMON block capabilities.

These structures make it possible to construct libraries of compiled IBM BASIC modules, with each module within a library consisting 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. Comprehensibility — Each subprogram or module has a specific task that it performs on a small number of parameters or common variables. This can make the overall program much easier to understand.

2. Independence — Because the communication paths among the subprograms are isolated and well-defined, a given subprogram is dependent on another subprogram only in a controlled manner. This allows the subprograms to be developed independently, even by different programmers. It also permits the reuse of subprograms across several related applications.

3. 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 can also enhance the overall flexibility of the application program by allowing it to be tailored for another system or user (for example, by recompiling a single module and relinking the program).

Libraries of subprograms can be structured in two different ways. First, a single module can contain all the subprograms that together perform some set of functions. The second method is to use a library of modules where each module usually contains only one subprogram. In either case, each subprogram linked must have a unique name. If two modules are invoked that have a subprogram with the same name, a Linker error is generated.

Using IBM Personal Computer ISAM Files

The Indexed Sequential Access Method (ISAM) is a library of subroutines that allows access to files both sequentially and by index. The files used with ISAM have a special format.

Normally, BASIC supports two types of data files; sequential files and random access files. These two types are sufficient for most applications. *The major limitation of these two file types, however, is that they do not allow you to access the records in the file according to their content.* You must either search the entire file until the desired record is found, or, you must know the number of the desired record.

With the addition of ISAM, *you can access data based on the content of your records.* If, for example, 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 on the price and availability of marble, ISAM provides a fast way to do so.

Before discussing ISAM in detail, a few basic terms need to be defined.

DATA TYPE — A data type is 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.

DATA RECORD — A data record is the basic unit of data in an ISAM file. Most ISAM subroutines operate on one record at a time. Generally, there will be a record in your ISAM file for each employee, spare part, or whatever information you are storing. An ISAM file can contain any number of records.

FIELD — A field is part of a data record. It contains 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 — A key field is a special kind of field. It contains the value that ISAM uses to determine where a record goes in the file.

KEY VALUE — A key value is the value stored in a key field. For example, the key field might be 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.

Each ISAM file is physically two files: a data file and a key file. A key is a data field that has been identified and described to ISAM. Using ISAM, you can access records in the data file according to the value contained in the key field. Both files must be present to use ISAM.

File names cannot exceed 8 characters. Conventionally, data file names end with a .DAT extension and key file names end with a .KEY extension.

The data file consists of data records and, usually, a data dictionary. The data dictionary, which resides 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 the same value in a given field of more than one record. This information is stored in the data dictionary.

The key file contains the indexing information that ISAM uses to access the data in the data file. ISAM gets this information (where the fields are, what type of data they contain) from the data dictionary, in the data file. The indexing is in the form of B-trees. B-trees are a special kind of index that points to the records in the data file. There is one tree in the key file for each key that you specify.

Whenever you access an ISAM file, ISAM automatically obtains the information it needs from 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 information to the key file.

When choosing the key for a file, it is a good idea to make the value of that key unique for each record in the file. This may help to avoid any confusion when searching the records in the file for a particular key value. For example, social security number is a good field to use as a key because everyone has a unique social security number.

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 nonadjacent fields of the same or different data types, and may or may not be keys themselves. Split keys are explained further in the Split Keys section of this document.

There are two types of ISAM data records: nonsegmented and segmented. Record types cannot be mixed in one file.

Nonsegmented records are the type most often used. They contain key fields that have fixed sizes. They may, however, contain one field that is variable in length, as long as that field is the last field in the record.

The second type of record, called a segmented record, supports 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—It is strongly recommended that you use segmented records only if it is very important to minimize the amount of storage space used for variable-length fields.

Segmented records are further described in the Segmented Records section later in this Proceedings.

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 random I/O procedures. 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—The demonstration program included on the ISAM diskette, MAIL.BAS, is designed as an aid in developing your own ISAM programs.

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

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

Installing ISAM.EXE

ISAM.EXE is a file containing the assembly language subroutines that make up ISAM. It must be loaded into memory before you can call any ISAM subroutines. If you try to use any ISAM subroutine and ISAM is not loaded, an error code of 27 is returned.

To install ISAM in memory, enter:

```
ISAM
```

The following message appears on the screen:

```
Installing IBM Personal Computer ISAM
(C)Copyright IBM Corp 1984, 1985 Version 2.00
(C)Copyright Microsoft Corp 1984, 1985
```

Because the ISAM routines take up memory space, you may want to remove them from memory when your program is finished. To do this, enter:

```
ISAM /F
```

No message appears when ISAM is removed.

You can also control the amount of buffer space that the ISAM routines can use while your program is running. To do this, entering the following when you install ISAM:

```
ISAM /S:xxxxx
```

xxxxx is the number of bytes of buffer space to be allocated. This number can range from 10000 to 65536.

If you do not use the **/S** option or you specify a number less than 10000, the buffer space defaults to 10000. ISAM performance improves as buffer space increases; however, this decreases the amount of memory available to your BASIC compiled program.

ISAM Terms and Concepts

This section explains some of the terms and concepts used in ISAM.

File Handles

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

Data Records

ISAM files are similar, in some ways, to BASIC random files.

When using random files in BASIC, the FIELD statement is used to allocate buffer space for the output data. This data buffer contains the variables that make up the data records in the file. For example:

```
FIELD #1, 1Ø AS NAME$, 25 AS ADDR$
```

This establishes a data buffer of 35 bytes; 10 for NAME$ and 25 for ADDR$. Because the space for this data buffer is allocated at compile time, NAME$ and ADDR$ are stored consecutively in memory.

To write a record to the file, you store the proper values into NAME$ and ADDR$, (using LSET and RSET), and write the buffer to the file. To write another record, you change the values in the data buffer and write it out again to the file.

When using ISAM files, you must also establish a data buffer. Two methods are given here to accomplish this:

- Using the FIELD statement
- Using the COMMON statement.

Using the FIELD Statement: The following example establishes the same buffer for an ISAM file:

```
OPEN "NUL" AS #9 LEN=35
FIELD #9, 1Ø 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.

To get the pointer to this data record, you can use the following procedure:

1. Insert the following function definition into your program.

   ```
   DEF FNSADD!(VPTR)
   FNSADD!=PEEK(VPTR+3)*256.Ø + PEEK(VPTR+2)
   END DEF
   ```

   This function accepts the pointer to a string descriptor as input and returns the address of the string. You must include this function in your program if you use the FIELD statement to establish the data buffer.

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.

Key Handles

The key handle is a 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.

Split Keys

A split key is a key that is made up of more than one field. The component fields of a split key may or may not be adjacent; may be the same or different data types; and may be nonkey fields, keys or split keys. All the components of one split key must have the same key handle.

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 one 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.

Split keys cannot be used with segmented records.

Segmented Records

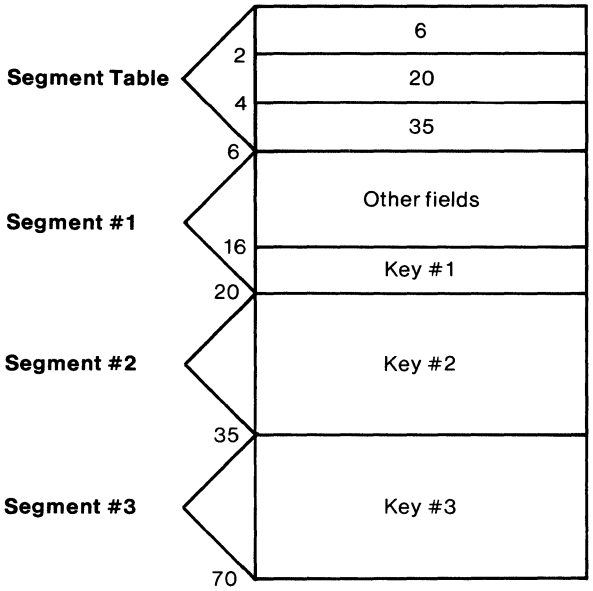
ISAM data files contain either segmented or nonsegmented records. These record types cannot be mixed in one file.

The address of a key field is given by a segment number and an offset. For nonsegmented 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 *n*th 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.

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, it is *strongly* recommended that segmented records be used 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 with 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.

Segmented records cannot contain split keys.

Record Description

The record description tells how many keys are in the record, if the record is segmented or nonsegmented and the minimum record allocation. This information is given to ISAM as the array Rdes.

Rdes(1) = *number of fields*

The number of fields declared in the field descriptor array.

Note—This includes both key and nonkey fields and components of split keys.

Rdes(2) = *segment-flag*

If the record is nonsegmented, *segment-flag* = 0. If the record is segmented, *segment-flag* =1.

Rdes(3) = *minimum record allocation*

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

It is recommended that you 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*

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*

A reserved word area. It must be initialized to zero.

Kdes(3) = *data-type*

The data type of the field. The data type is set by supplying one of the following words: **INTEGER, STRING, NUMERIC, SINGLE, DOUBLE.**

Note—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 nonsegmented 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*

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.

Note—If you are defining a nonkey field, this number is 0.

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

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

duplicates-allowed flag

A flag indicating whether duplicate values are allowed for this particular field.

descending-flag

A flag inverting 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.

case-insensitive-flag

A flag causing string-based data types to ignore differences in case (for example, the values 'FiRst' and 'first' would be equal).

Kdes(8) [low byte] = *field-mode*

Tells if the field is a key. If it is a key, it tells if it is a split key.

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

Kdes(9) = *filler*

A reserved word area. It must be initialized to zero. The following example shows typical record and key descriptors:

```
DIM RDES(3), KDES(9)
'$INCLUDE:'ISAM.INC'
RDES(1) = 1                     '1 Key field
RDES(2) = Ø                     'Nonsegmented
RDES(3) = 8                     'Minimum record length
FIELDNAME$="ID NUMBER"
KDES(1) = VARPTR(FIELDNAME$)    'Field name pointer
KDES(2) = Ø                     '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) = Ø                     'Reserved-Set to zero
```

Examples

This example shows how to use a string variable as a key:

```
' FILENAME = EX3.BAS

'$LINESIZE: 132

DEFINT A-Z

DIM RDES(3)                          'Record Descriptor
DIM KDES(9)                          'Key Descriptor

COMMON SHARED /ISAM/ IXSTAT,IOSTAT   'Isam Status Variables

COMMON /DATA.RECORD/ ENAME$,ID$

'$INCLUDE:'ISAM.INC'

DEF FNSADD!(VARPOINTER)                  ' get the address of a string
                                         ' given the varpointer of the string
  FNSADD! = PEEK(VARPOINTER + 3)*256.0 + PEEK(VARPOINTER + 2)

END DEF
'

OPEN "NUL" AS #9 LEN=27              'Open needed for FIELD

FIELD #9,
    2Ø AS ENAME$,                    'Key field
     7 AS ID$                        'ID field

' Below we set up the record descriptor

RDES(1) = 1                          '1 field declared
RDES(2) = Ø                          'Not segmented
RDES(3) = Ø                          'Default record allocation

FIELDNAME$ = "EMPLOYEE NAME"         'Field Name

' Below we set up the key descriptor

KDES(1) = VARPTR(FIELDNAME$)         'Field name pointer - first key
KDES(2) = Ø                          'Reserved - Set to Ø
KDES(3) = STRING                     'Data type - String
KDES(4) = 1                          'Segment 1
KDES(5) = 1                          'Position 1
KDES(6) = 2Ø                         'Length of field
KDES(7) = 1                          'Key number is 1
KDES(8) = (Ø * 256) + 1              'Single, nonsplit key
KDES(9) = Ø                          'Reserved - Set to Ø
```

```
FN$ = "EX3.DAT"                          'Data file name

PRECDESC! = VARPTR(RDES(1))              'Pointer to the record descriptor
PKEYDESC! = VARPTR(KDES(1))              'Pointer to the key descriptor
PDATDESC! = FNSADD!(VARPTR(ENAME$))      'Pointer to the data descriptor

KEYNUM = 1
KEYLEN = 2Ø
RECSIZE = 27                             'Length of data record

' Below we open the file in create mode

CALL IOPEN(VARPTR(FN$), 3, PRECDESC!, PKEYDESC!, FILENUM)

IF IXSTAT <> Ø THEN PRINT "OPEN FAILED":STOP

INPUT "ENTER EMPLOYEE NAME AND ID # (Ø TO QUIT)";NAME$,I$
WHILE VAL(I$) <> Ø
  LSET ENAME$ = NAME$
  LSET ID$ = I$
  CALL IWRITE(FILENUM,PDATDESC!,RECSIZE)
  IF IXSTAT = 13 THEN PRINT "RECORD NOT WRITTEN - DUPLICATE KEY"
  PRINT
  INPUT "ENTER EMPLOYEE NAME AND ID # (Ø TO QUIT)";NAME$,I$
WEND

CALL ICLOSE(FILENUM)
IF IXSTAT <> Ø THEN PRINT "AFTER CLOSE, IXSTAT =";IXSTAT:GOTO DONE

CALL IOPEN(VARPTR(FN$), 2, PRECDESC!, PKEYDESC!, FILENUM)
IF IXSTAT <> Ø THEN PRINT "AFTER OPEN, IXSTAT =";IXSTAT:GOTO DONE

PRINT
INPUT "ENTER EMPLOYEE NAME FOR RECORD YOU WANT TO CHANGE";
      NAME$
LSET ENAME$ = NAME$

CALL ISEEK(FILENUM, KEYNUM, PDATDESC!, KEYLEN, 2) ' GET EQUAL RECORD
IF IXSTAT = 1Ø THEN PRINT ENAME$;"NOT IN THE FILE":GOTO DONE
IF IXSTAT <> Ø THEN PRINT "SEEK FAILED - IXSTAT = ";IXSTAT:GOTO DONE

PRINT
INPUT "ENTER NEW EMPLOYEE NAME AND ID NUMBER";NAME$,I$
LSET ID$ = I$
LSET ENAME$ = NAME$
CALL IREWRITE(FILENUM,PDATDESC!,RECSIZE)

IF IXSTAT <> Ø THEN PRINT "AFTER IREWRITE,IXSTAT = ";IXSTAT

DONE:
CALL ICLOSE(FILENUM)
IF IXSTAT <> Ø THEN PRINT "AFTER CLOSE #2, IXSTAT =";IXSTAT

CALL ICLOSE(FILENUM)

END
```

This program opens the file created with the previous example and prints out all the data records.

Note that we don't have to assign values to the record and key descriptors, because we are using a file that has already been created.

```
' FILENAME = EX4.BAS

'$LINESIZE: 132

DEFINT A-Z

COMMON SHARED /ISAM/ IXSTAT,IOSTAT     'Isam Status Variables

Def FNsadd!(varpointer)                    ' get the address of a string
                                          ' given the varpointer of the string
  FNsadd! = peek(varpointer + 3)*256.Ø + peek(varpointer + 2)

end def

Open "nul" as #9 Len=27                'Open needed for FIELD

Field #9,
    2Ø as ENAME$,                      'Key field
     7 as ID$                          'ID field

FN$ = "EX3.DAT"                        'Data file name
NULL! = Ø

PRECDESC! = VARPTR(NULL!)
PKEYDESC! = NULL!
PDATDESC! = FNSADD!(VARPTR(ENAME$))    'Pointer to the data descriptor

KEYLEN = 2Ø

RECSIZE = 27                           'Length of data record

' Below we open the file in read/write mode

CALL IOPEN(VARPTR(FN$), 2, PRECDESC!, PKEYDESC!, FILENUM)
IF IXSTAT <> Ø THEN PRINT "AFTER OPEN #3, IXSTAT =";IXSTAT:GOTO DONE

PRINT:PRINT "THIS IS THE ENTIRE FILE :":PRINT
CALL ISEEK(FILENUM, KEYNUM, PDATDESC!, KEYLEN, Ø) ' GET EQUAL RECORD
AGAIN:
CALL IREAD (FILENUM,PDATDESC!,RECSIZE)
IF IXSTAT <> Ø THEN PRINT "READ FAILED - IXSTAT =";IXSTAT
PRINT ENAME$,ID$
CALL INEXT(FILENUM,KEYNUM)
IF IXSTAT = Ø THEN GOTO AGAIN  ' else the file is done

DONE:

CALL ICLOSE(FILENUM)

END
```

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*]][;]]

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:

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

– Erase an object module.

* Retrieve an object module and copy 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:

–+ Replace an object module with the contents of the object file of the same name (plus .OBJ).

–* Remove an object module and at the same time erase 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.

The operations are performed in the following order:

1. Erasures and removals
2. Additions

Erasures and removals are performed in the order in which the specified object modules occur in the library. Additions are performed in the order you specify.

Examples:

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;
```

Note that the following also have the same effect:

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

LIB BASIC+TEST-TEST;
```

If you want to make the same change but put the changes in a new library called BASNEW.LIB, any of the following work:

```
LIB BASIC-+TEST,,BASNEW

LIB BASIC-TEST+TEST,,BASNEW

LIB BASIC+TEST-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.

The IBM Personal Computer Technical Reference Manuals

The IBM Personal Computer Technical Reference Manuals were redesigned after publication of the revised manuals for the IBM PC and the IBM PCXT in June 1984.

Common pages covering I/O devices and adapters were removed from each manual and placed in an **Options and Adapters Reference Manual (OARM).** Now there is only system information in each manual, thus creating a System Technical Reference Manual.

This set of technical reference manuals consists of two publications:

- One of the following System Technical Reference Manuals (STRM):
  - 6322507 IBM Personal Computer STRM or
  - 6322508 IBM Personal Computer XT and IBM Portable Personal Computer STRM

  AND

- The Technical Reference Options and Adapters, Volumes 1 and 2
  - 6322509 OARM

A subscription service (Update Information Service) is included with the Options and Adapters Reference Manual. An enrollment card is placed under the shrink wrap of the manual. When the card is filled out and sent to IBM, technical information updates are provided through June 30, 1985. In August 1984, the PCAT STRM (1502243) IBM Personal Computer AT System Technical Reference Manual was also added to this subscripton service.

Card Design Guideline

Advances in technology make it difficult to expect items designed today to be totally adaptable to future systems.

Consider the advances in system processors which have gone from 8-bit to 16-bit. The increased capability allows wider data paths and results in continuing changes in system bus architecture and physical layout.

With Very Large Scale Integration (VLSI) technology, system units can be made even smaller, and smaller units could restrict the physical size of the adapter cards. This could require the cards to be re-mapped for a future system. Processor speed is another important consideration. Adapter cards designed for slower processors may not run on faster processors and vice versa because of timing considerations.

There are, however, some general criteria which can be followed in designing adapter cards to operate across the family of IBM Personal Computers:

- 8-bit bus architecture *
- PC/PCXT physical card dimensions *
- Do not extend card area below the top of the bus connector

Cards designed according to these criteria will operate in the IBM PC, the IBM PC XT and the IBM Personal Computer AT, except those cards which have unique timing considerations.

If a developer wishes to take advantage of the IBM Personal Computer bus, then change the bus design but leave the other parameters the same. If more space is needed than the PC/PCXT card dimension allows, then a larger form factor is allowed for the IBM Personal Computer AT. This, however, will make the adapter card usable only in the IBM Personal Computer AT.

* Adapter data available in technical reference manuals

IBM Personal Computer Seminar Proceedings

Volumes and Topics

V1.1 Contains identical information as V1.2

V1.2 DOS 2.0 and 1.1 Comparison
Compatibility Guidelines for Application Development
8087 Math Co-Processor
IBM Macro Assembler

V1.3 DOS 2.1, 2.0 and 1.1 Comparison
Disk Operation System 2.1
IBM PC*jr* Architecture
IBM PC*jr* Compatibility Overview
Cartridge BASIC
IBM Personal Communications Manager-Modem Drivers

V2.1 IBM Software Support Center
International Compatibility Requirements

V2.2 IBM Software Support Center
International Compatibility Requirements
IBM Personal Computer Cluster Program

V2.3 IBM Personal Computer Cluster Program
Sort, Version 1.00
FORTRAN Compiler, Version 2.00
Pascal Compiler Tips and Techniques

V2.4 IBM Personal Computer AT Architecture
ROM BIOS Compatibility
DOS 3.0
Software Compatibility

V2.5 IBM PC Network Overview
IBM PC Network Hardware
IBM PC Network BIOS (NETBIOS) Architecture
IBM PC Network Program

V2.6 TopView

V2.7 IBM Personal Computer Resident Debug Tool

V2.8 IBM PC Network SMB Protocol

V2.9 IBM PC Xenix

V2.10 IBM PC Professional Graphics Software
IBM PC Graphical Kernel System
IBM PC Graphical File System
IBM Plotting System Library
IBM Professional FORTRAN
IBM PC Data Acquisition and Control Adapter and Software
IBM General Purpose Interface Bus Adapter and Software

V2.11 IBM Enhanced Graphics Adapter

V3.1 IBM PC Information Panel (3295 Plasma Display)

V3.2 IBM BASIC Compiler 2.00

Notes

IBM Corporation
Editor, IBM Personal Computer Seminar Proceedings
4629
Post Office Box 1328
Boca Raton FL 33432