BASIC/Z

native code compiler

System/z, inc.

Table of Contents

Table of Contents

EXECUTIVE COMMANDS

Table of Contents

RESERVED WORDS

Table of Contents

RESERVED WORDS

Table of Contents

RESERVED WORDS

Table of Contents

RESERVED WORDS

Table of Contents

RESERVED WORDS

USER-DEFINED FUNCTIONS

FILE HANDLING

ERROR HANDLING

Table of Contents

'ENDICES

BASIC/Z

native code compiler

reference manual

revision 1.11: October, 1983
cpu type: 8080/8085/Z80
operating system: cp/m, mp/m, turbodos

401 - 450

System/z, inc.

System/z, inc.

P.O. Box 11
Richton Park, IL 60471
(312) 481-8085

PROPRIETARY NOTICE:

Operating systems supported by this version of BASIC/Z:

CP/M - rev #2.X
CP/M Plus - rev #3.X
CP/M-86/80 - rev #1.X (DEC)
CP/M-86/80 - rev #2.X (DEC)
EXTENDED CP/M - rev #2.X (Vector Graphic)
MP/M II - rev #1.X
TURBODOS - rev #1.1X
TURBODOS - rev #1.2X

BASIC/Z - not your basic BASIC

BASIC/Z is an extended implementation of the BASIC language. As a native code compiler, it generates executable machine code which is compatible with 8080, 8085, and Z-80 microprocessors. It requires a 48K or larger system, as well as a console device with addressable cursor. BASIC/Z supports numerous single user and multi-user operating systems, including CP/M (2.x/3.0x), MP/M II, Turbodos, Vector Graphic Extended CP/M, and other similar environments.

BASIC/Z offers many unique features never before available to the microcomputer user. As an interactive compiler, it includes a full function program editor with advanced features such as global search and change, sixteen local edit commands, and much more. An extended debugging facility is provided, which even allows "single-stepping" of a compiled program, with continuous display of selected variables. Best of all, each program line is thoroughly tested for proper syntax at the time it is typed !

We hope BASIC/Z will prove to be a valuable addition to your software library. We welcome your comments, suggestions, and your criticisms, as they help us to provide you with a better product. For just this purpose, postage-paid user comment forms are included at the end of this reference manual.

Please note that the End User License Agreement only allows usage of the BASIC/Z compiler on a single computer system. It may not be distributed in any form whatever. If you wish to execute BASIC/Z on more than one computer system, secondary cpu licenses are available at a substantial discount from the single system price. Also, multi-copy OEM licenses are available for all System/z software products. Please contact System/z, inc. for further details.

Software License Agreement

Enclosed with BASIC/Z is a copy of the End User License Agreement. Before you do anything else, it is essential that you read it most carefully. All programs and data recorded on the master disk are copyrighted, and may be used and copied only under the terms and conditions of the agreement. Any unauthorized reproduction, transfer, or use of these materials may be a criminal offense under Federal or State law. It should be specifically noted that any form of rental, loan, temporary transfer, or trial sale allowing usage of the BASIC/Z master programs is expressly prohibited.

You are welcome to review the BASIC/Z documentation and license agreement at no risk for up to 21 days. Your software dealer will gladly refund your full purchase price if the master disk, manual, license agreement, and proof of purchase are returned, in new condition, within 21 days of the purchase date. However, absolutely no disk may be returned for exchange, refund, nor credit once the sealed master disk envelope has been opened. A defective disk will be replaced with the identical title and format only.

Please be certain to completely fill out the license agreement, and return it to System/z, inc. at the earliest possible date. Your master disk includes an update utility which allows you to avail yourself of the continued improvement and upgrading of BASIC/Z. However, your completed license agreement must be received by System/z, inc. within 21 days of the purchase date to qualify for this free service.

Also, until the agreement is completed and returned, no permission is given to use any of the supplied programs or data, nor is any authority granted to distribute any object program compiled using BASIC/Z. Absolutely no software support (including corrections, updates, bulletins, etc.) will given by System/z, inc., or its representatives, to an unlicensed user.

In much of our literature and documentation, including this very manual, references will be made to certain terms and conditions of the license agreement, and both the conditions and the rights which it grants or imposes. That information is supplied in order to further explain or illustrate terms of the agreement. However, no such literature should be construed as either extending or modifying the basic agreement. For precise information, you must refer to your copy of the End User License Agreement.

continued - -

Software License Agreement (continued)

BASIC/Z and the associated support programs are distributed by System/z, inc., and by dealers licensed by System/z, inc., to distribute BASIC/Z. A disk which contains an End-User Master copy of BASIC/Z has a label affixed to it, as shown below:

System/z, Inc.
PO Box 11
Richton Park, IL 60471
(312) 481-8085

Title _____ Serial #_____

Rev. #_____ System _____ Format _____

This software is serialized, and may be used only by the licensee, on one registered computer. It may not be rented, loaned, resold, transferred, nor assigned, without the written consent of System/z, inc. All software provided is copyrighted, and may be used or copied only under the terms and conditions of the accompanying End User License Agreement. Unauthorized reproduction, transfer, or use of this material may be a criminal offense under Federal or State law.

End User Master Disk

Another type of disk label, that of a SECONDARY SUB-LICENSE, is shown below. This is not a master disk, and may not be sold, transferred, nor assigned, in any manner, unless it accompanies an End-User Master Disk, as shown above. This type of disk is distributed only to users wishing to license one or more additional systems, after they have first purchased a primary copy of BASIC/Z. These added restrictions must be imposed due to the substantial reduction in price for multiple copies of BASIC/Z.

System/z, Inc.
PO Box 11
Richton Park, IL 60471
(312) 481-8085

BASIC/Z Serial # _____

Rev. #_____ System _____ Format _____

This software is serialized, and may be used only by the licensee, on one registered computer. It may not be rented, loaned, resold, transferred, nor assigned, without the written consent of System/z, inc. All software provided is copyrighted, and may be used or copied only under the terms and conditions of the BASIC/Z End User License Agreement. Unauthorized reproduction, transfer, or use of this material may be a criminal offense under Federal or State law.

Secondary Sub-License - Not a Master

A third type of disk label, that of an OEM END-USER MASTER DISK, is shown below. This type of disk is distributed only to users who have received their copy of BASIC/Z "bundled" with a new computer system, and supplied at no specific extra charge by the manufacturer of the computer. This copy of BASIC/Z is licensed for use only on the one computer with which it was supplied. It may not be sold, transferred, nor assigned, in any manner, unless it accompanies the registered computer.

System/z, Inc.
PO Box 11
Richton Park, IL 60471
(312) 481-8085

BASIC/Z Serial # _____

Rev. #_____ System _____ Format _____

This software is serialized, for use only on the computer with which it was supplied. It may not be rented, loaned, sold, transferred, nor assigned unless it accompanies the registered computer. All software provided is copyrighted, and may be used or copied only under the terms and conditions of the BASIC/Z OEM End User License Agreement. Unauthorized reproduction, transfer, or use of this material may be a criminal offense under Federal or State law.

OEM End User Disk

A BASIC/Z master disk is delivered in a sealed envelope, with the following label affixed to the outside of the envelope:

System/z, inc.
P.O. Box 11
Richton Park, IL 60471
(312) 481-8085

BASIC/Z Serial# _____

Rev.# _____ System _____ Format _____

— MASTER DISK ENCLOSED —

— DELAY OPENING! —

Examine the license agreement and documentation most carefully. The act of opening this sealed envelope constitutes your acceptance of all terms and conditions of the accompanying license agreement. Due to federal laws pertaining to potential copyright infringement, no disk may be returned for exchange, refund, nor credit once this sealed envelope has been opened. A defective disk will be replaced with the identical title and format only.

If your disk was not delivered in a sealed envelope, or the disk or the envelope do not have labels affixed, as shown, or you did not receive a copy of the End User License Agreement, please contact System/z, inc. directly, as soon as possible.

1.001 A programmer's overview of BASIC/Z

BASIC/Z is a powerful tool for the experienced programmer. This reference manual assumes a working knowledge of the BASIC language, as well as the operating system under which BASIC/Z will run. No attempt will be made here to provide a "tutorial" on programming. Rather, special emphasis will be given to areas where BASIC/Z differs from other popular languages. Beginning programmers are urged to consult one of the many fine texts on elementary BASIC before attempting to utilize BASIC/Z.

BASIC/Z offers four unique numeric variable types. Control and control/d variables offer one and two byte unsigned binary integers. They are most frequently used as counters, pointers, etc. The remaining two types, integer and real (i.e. floating point), are stored in a bcd format which eliminates the round-off or conversion errors common to binary math systems. Also, they offer the unusual concept of program definable precision. The amount of memory used to store each real or integer variable may be defined with an optional SIZES statement, to yield from six (6) to eighteen (18) digits of accuracy for either integer or real calculations. The default size of a real variable (RSIZE) is five (5) bytes, offering eight (8) digits of precision, while the default size of an integer variable (ISIZE) is three (3) bytes, to yield six (6) digits of accuracy. Incidentally, all numeric functions in BASIC/Z (even trigonometric and logarithmic functions) will calculate an eighteen digit result accurately.

Unlike many other languages, BASIC/Z allocates string space statically. That is, the memory location of a string variable does not generally change during execution of a program, regardless of how many times the value is altered. This offers a substantial speed advantage, as "garbage-collection" delays are eliminated. However, it imposes an additional burden on the programmer, as every string variable (both scalar and array) must have a defined maximum length, which may not be exceeded.

BASIC/Z implies a default maximum string length (known as SSIZE) of forty (40) characters for any scalar (non-array) string variable. However, an optional SIZES statement may designate an explicit SSIZE, in the range of 1 through 250, which takes precedence over the implied value. Any scalar string may be explicitly declared to have a maximum length of other than SSIZE, by including it in a DIM or COMMON statement. Every string array must be explicitly declared with a DIM, DDIM, or COMMON statement, which includes a specific definition of the maximum length of each individual element of the array.

Unlike certain other languages, BASIC/Z does not include a statement such as LPRINT, to output directly to the printer. Rather, the concept of a single "print stream" is utilized. That is, all printed output is directed to this single stream with PRINT statements, regardless of the intended device. During program execution, various print re-direction statements may be used to dynamically define the destination device. For example, at start-up, all PRINT statements

continued - -

1.001 A programmer's overview of BASIC/Z (continued)

are directed to the console. If LPRINTER is executed, all subsequent PRINT statements will be directed to the printer. Likewise, SPOOL 21 will direct them to file #21, etc. This concept allows a single routine to handle output to several devices, with appropriate savings of system resources.

BASIC/Z includes a full function program editor, which is described later in this manual. Most programmers will choose to use it to create source programs because of the unusual features it provides, such as syntax testing as you type. However, it is possible to use any editor which creates standard ascii files in the format described in section 1.007, with just one additional procedure required.

BASIC/Z will only compile source programs which are saved in internal tokenized format, having an extension of .BZS. Therefore, your completed ascii source program must be saved with any extension except ".BZS". It is then loaded with the BASIC/Z editor, which will test the syntax of the entire program. If your editor does not require line numbers, the &L option to the load command should be specified, and BASIC/Z will insert them as necessary. The SEARCH command is used next to locate any single quote (') characters, which are inserted by BASIC/Z to flag any syntax errors. If none are found, the save command is used to save a tokenized version of the program which may then be compiled.

BASIC/Z allows precise control over all aspects of program error handling. For example, even running under standard cp/m, BDOS errors may be trapped as any other system error. Also, as many as 61 error traps mays be concurrently defined. In addition to a global trap, each of up to 30 files may optionally have an individual local error trap, and an end-file trap as well. A special point to note is the handling of subroutine, function, and expression stacks. Generally, in case of an error condition, all three stacks are reset. Unlike any other microcomputer language, this action may be suppressed, as needed, with the CLRNONE statement. This offers the unique ability to trap an error condition within a subroutine, or multi-line user-defined function, and still continue program flow through the normal RETURN or FNEND statements. This is an area which should be studied most carefully (see section 7), as failure to do so may well cause generation of additional errors such as "stack overflow" or "invalid return".

While this short summary cannot tell you everything about BASIC/Z, we feel confident that this manual, taken as a whole, will do just that. Read it thoroughly, and we think you will find BASIC/Z to be a powerful tool designed with you, the applications programmer, in mind. Some particular areas of close scrutiny should probably include: multi-line, user defined functions in section 5, screen handling functions such as TAB and EDIT$, file handling, which is overviewed in section 6, as well as error handling which is detailed in section 7 and appendix C.

1.002 Initializing the BASIC/Z system

Your BASIC/Z system includes a master disk, with serialized versions of the BASIC/Z compiler (BZ.COM and BZ.OVL), the RUN/Z run-time library (RZ.COM), and several other utility programs described elsewhere in this manual. Prior to any operation, we urge you to immediately copy all of the supplied programs, and store your BASIC/Z master diskette in a safe place for archive purposes only. Prior to copying, be certain to physically write-protect the master disk, to avoid any possibility of damage to it.

Both the BASIC/Z compiler and the RUN/Z run-time library must be configured to your individual requirements. This includes definition of the hardware environment (to allow cursor addressing, crt attributes, etc.), as well as just recording your personal preferences (such as the number of "form-feeds" to be issued after a printed compiler listing). Configuration is accomplished by executing a series of INSTALL programs provided on your master disk. A complete description of the INSTALL process will be found in appendix F of this manual.

A word of caution - never, under any circumstances, allow the programs on your master disk to be configured with INSTALL. As originally supplied to you, they are pre-configured to execute properly with any supported operating system. However, should you install them, it is possible you might be unable to execute INSTALL again, if you change to a new operating system! For safety, always execute INSTALL with an unconfigured version of RUN/Z.

1.003 Executing BASIC/Z

To begin execution of the BASIC/Z compiler, both BZ.COM and BZ.OVL must be present on your disk system. To execute, type ~

[dr:]BZ <RETURN>

BASIC/Z will now "sign-on" with the title, revision and serial number, etc., and await your command. You may now enter either executive commands or BASIC/Z program lines.

In order to use available system memory most efficiently, the actual compilation process of BASIC/Z runs in multiple overlays, each of which are loaded from disk automatically. Therefore, the disk from which BZ.COM and BZ.OVL were originally loaded may not be removed, to assure proper execution.

BASIC/Z will recognize an implied form of the LOAD command. At the time of execution, the name of a source program to be loaded may be included as a command line trailer. For example, a command line of: 'BZ PROGRAM1' would cause BASIC/Z to be executed, followed by an automatic load of the source program 'PROGRAM1.BZS'. Similarly, during program testing and debugging, the final executable statement in your program might be: LINK "BZ", "PROGRAM1" for the same result.

1.004 Entering executive commands

An executive command specifies an operation to be performed from the command level of BASIC/Z. It consists of the "command word", optionally followed by one or more parameters. If a command line contains more than one parameter, each must be separated by a valid delimiter, a comma or a space. In command line entry, BASIC/Z will recognize the standard back-space and control-x (erase entire line) functions.

Executive commands may not be inserted into BASIC/Z programs. They are designed to be used interactively, from the command level of BASIC/Z only. A detailed description of the available executive commands will be found in section 3 of this manual.

1.005 Entering BASIC/Z program lines

Each line in a BASIC/Z program must begin with a valid line number in the range of 0 through 65535. This line number is used to determine the physical placement of the line within the program, and may optionally be used as the argument to certain BASIC/Z statements, such as GOTO, DEBUG, etc. This line number may be manually typed, or BASIC/Z will assume this task for you with automatic line number increments of your choice. If the space bar is pressed as the first character of a line, BASIC/Z will auto-generate an appropriate line number, as described under the AUTO executive command in section 3.003 of this reference manual. In program line entry, BASIC/Z will recognize the standard back-space and control-x (erase entire line) functions.

Entry of a program line is terminated by pressing <RETURN>. At that time, BASIC/Z will thoroughly test it for proper syntax. If the line is found to be syntactically perfect, it will be inserted into the program buffer (but any prior line with the identical line number will be over-written!). If any syntax errors are found, an appropriate message will be displayed, along with a "^" pointing to the first syntax error in the line. The EDIT mode will be entered, so that all of the special edit commands described in section 3.017 will be available to assist you in correction of the error. The EDIT mode may be aborted at any time with the special edit command Q (quit), but this will cause the program line under edit to be discarded.

Program lines may be erased with the executive command DELETE, which is described in section 3.012 of this manual. BASIC/Z will also recognize an implied form of the delete command. Enter just a line number, followed by pressing <RETURN>, to erase a single line from the program buffer.

1.006 Terminating BASIC/Z operation

Operation of BASIC/Z may be terminated with the executive command DOS. Upon execution, control is returned to the operating system.

1.007 BASIC/Z program files

SOURCE: A BASIC/Z source file is the English-like representation of a program, in the format it was typed with the editor. It is stored on disk in a specially compressed, internal format, which is noted by a file type of ".BZS". In this standard source format, all reserved words are replaced by special tokens, and BASIC/Z is able to presume that the entire program is syntactically perfect. Only source programs which are saved in this standard format may be compiled.

ASCII: BASIC/Z will also load, save, and merge source programs in ascii format, which are stored on disk with any extension other than ".BZS". In this format, all tokens and line numbers are expanded to ascii. Each line is terminated by a cr/lf, and end-of-file is signified by a control-z (1A hex). This is allowed to facilitate loading of programs created for other languages, or with other program editors, as well as from different revisions of BASIC/Z, which may not be syntactically compatible. As each line of an ascii file is loaded by BASIC/Z, it is tested for proper syntax. If a valid line number is not present, the entire line must be discarded (unless the &L option has been invoked in the load command). If any syntax errors are found, the line will be converted to a remark by inserting a single quote (') at the first position of the line. Any such errors may then be located with the search command, and corrected.

OBJECT: When a source program is completed, BASIC/Z may be used to compile it into machine code form that the computer can actually execute. This is called an object file, designed only for the computer to read, at "run-time". BASIC/Z compiled object programs are stored on disk with an extension of ".BZO", and may be immediately executed with the RUN/Z run-time library. Optionally, the BIND command (section 3.006) may be invoked to combine the object program and needed support routines into a single executable ".COM" file.

1.008 Creating a BASIC/Z program

A new BASIC/Z program is created by typing one or more program lines, as described in section 1.005. It must be given a unique name with the TITLE command (section 3.042), and stored on disk as a source file with the command SAVE (section 3.037).

Compilation is the process by which your BASIC/Z source program is translated into machine code which may be executed directly by the microprocessor. This process is initiated by the COMPILE command, as described in section 3.010.

1.009 Executing a BASIC/Z program

Once a BASIC/Z source program has been compiled into an object program, it may be executed by invoking the RUN/Z run-time library. From the operating system level, this is accomplished by typing ^

```
RZ  filename  [&command]
```

From the command level of BASIC/Z, it is accomplished by typing ^

```
EXEC  filename  [&command]
```

As the filename argument is assumed to be a compiled object program, it is not necessary to include the extension of ".BZO" in either of the above cases.

As an option, the BIND command (section 3.006) may be invoked to combine the object program and needed support routines into a single executable ".COM" file. From the operating system level, the command file is executed by typing ^

```
filename  [&command]
```

From the command level of BASIC/Z, it is executed by typing ^

```
LINK  filename  [&command]
```

As the filename argument is assumed to be a command program, it is not necessary to include the extension of ".COM" in either of the above cases.

In all four of the above examples, the &command parameter is optional. If it is present, it may be retrieved with the function COMMAND$ (section 4.022). This feature could allow you to "password-protect" a BASIC/Z program. The &command parameter may optionally begin with an ampersand (&), but this acts as a delimiter only, and is never passed to the function COMMAND$.

1.010 Interrupting a BASIC/Z program

Except for the DEBUG mode (section 4.036), an executing BASIC/Z program may not be interrupted by console entry of a control-c, nor paused with control-s. This may help to prevent an unsophisticated user from inadvertantly "crashing" a critical program. However, BASIC/Z does provide the full facility to simulate this function.

```
IF INCHAR$ = CHAR$(3) THEN GOTO @CANCEL.ROUTINE
```

In the above example, control would be transferred to the routine labeled @CANCEL.ROUTINE if a control-c were entered from the console. In this way, an executing program can be interrupted on a "controlled" basis, with proper closing of open files, etc.

1.011 Command level console output

At the command level of BASIC/Z, all output to the system console is handled by our unique "interruptable speed control". Listings may be displayed at any of nine user-selectable speeds, which may be altered at will, from the keyboard, without halting execution. This allows a quick scan to a needed section, with subsequent "slow-stepped" output, to allow for careful review.

Initially, all data to the console is output at a middle range speed. It may be altered at any time by merely depressing a key from "1" through "9". Entry of a "1" will cause data to be sent at the maximum possible speed, while each higher number will cause the output to become progressively slower.

Entry of control-c or <RETURN> will cause termination of the display. Entry of any other key will cause BASIC/Z to pause. Display may be restarted by depressing any key other than a control-c or <RETURN>.

1.012 BASIC/Z revision numbers

Every BASIC/Z source and object program is encoded with the revision number of the compiler which created it. This is done to prevent difficulties if it becomes necessary to change any syntax requirements for BASIC/Z in the future. The current revision number of your compiler is displayed at "sign-on", and may also be found on the label of your master disk.

The major revision number is considered to be that to the left of the decimal point, while the minor revision number is that to the right of the decimal point. In order for a source program to be loaded or compiled by BASIC/Z, the major revision number must be identical, and the minor revision number must be equal or smaller. In order for a compiled object program to be executed with RUN/Z, both the major and minor revision numbers must be identical.

In the event of upgrade to an incompatible major revision, a source program would merely be saved in ascii format with the older version of BASIC/Z and re-loaded and saved with the newer version. In this way, any potential syntax problems are automatically "flagged" by BASIC/Z. The updated source program may then be re-compiled for execution with the later revision of RUN/Z.

1.013 Distributing programs compiled with BASIC/Z

BASIC/Z is the language of choice, considering distribution of compiled application programs. It requires absolutely no royalty payments, and is readily adaptable to almost any combination of terminals and printers, without requiring a new compilation.

The BASIC/Z End User License Agreement specifies all of the rights granted, as well as the conditions imposed, in using BASIC/Z. While many of these will be summarized here, nothing should be taken as modifying the terms of that agreement.

First and foremost, the BASIC/Z compiler (BZ.COM and BZ.OVL) may not be distributed, in any form, or under any conditions whatsoever. Likewise, the RUN/Z run-time library (RZ.COM) may not be distributed in a stand-alone form.

Compiled object programs may be distributed to any person or organization, at your sole discretion, for execution on as many computer systems as you see fit to allow. Generally, these compiled programs will have an extension of .BZO. However, as the RUN/Z run-time library may not be distributed as a stand-alone file, it is mandatory that support routines from it be integrated into at least one command (.COM) file with the BIND executive command.

While we grant you the right to distribute certain run-time routines (when integrated into a command file), no inference should be drawn that any of this code is placed in the public domain. All code comprising the RUN/Z run-time library is copyright (c) by System/z, inc., and you are required to protect our proprietary interest. The following statement must be published in documentation accompanying distributed programs:

> One or more of the supplied programs were compiled with the BASIC/Z compiler, a product of System/z, inc., of Richton Park, IL USA. Portions of these programs are proprietary to and copyright by System/z, inc., and may not be used, in whole or in part, for any other purpose. All rights are reserved.

Upon program execution, our copyright notice is displayed for approximately one-half second. While the screen may be erased immediately thereafter by your program, nothing may be done to inhibit display of this notification.

continued - -

1.013 Distributing programs compiled with BASIC/Z (continued)

When distributing programs, you may find it prudent to pre-install the package for your customer's system, as this may inhibit further copying. However, we grant you the authority to distribute INSTALL.COM, INSTALL1.BZO, INSTALL2.BZO, and INSTALL.DAT so that your program may be re-installed for other hardware. It is not necessary to provide INSTALL3.BZO to your customers, as it is used solely to install the BASIC/Z compiler.

Since the RUN/Z run-time library (RZ.COM) may not be distributed as a stand-alone program, you must use the executive command BIND to create command files for both INSTALL, and your primary application program. Some care must be exercised here to ensure compatibility of the chosen operating system, with that of the destination computer. Failure to do so could render the programs inoperative on your customer's computer. If you are pre-installing programs, your choice will be dicatated by the destination computer. If not, be certain that the command files you create with BIND utilize an unconfigured version of the run-time library. If there is any question of this, RUN/Z may be re-installed using "Non-specific operating system" as the operating system choice, to return it to an unconfigured condition. Failure to do so could be disastrous, due to incompatibilities in error handling, etc.

For certain commercial applications, the ability to modify the INSTALL programs may prove invaluable, as they could be customized to a special need. Also, additional default conditions could be established for your programs, through added installation parameters. Source code for the INSTALL programs, and the configurable data area of RUN/Z, is available from System/z, inc., at a moderate license fee. As limited support is provided, this should only be considered by highly experienced programmers.

2,001 Line format

Program lines in a BASIC/Z program have the following format:

lnum stmt { : stmt } <RETURN>

A program line in BASIC/Z must always begin with a line number, and may contain a maximum of 250 characters, including the digits in the line number. A line may include multiple statements, but each must be separated by a colon (:). Entry of a program line is terminated by pressing <RETURN>.

All BASIC/Z "reserved-words", as well as variable, label, and function names, must be immediately followed by a space if the syntax of the statement does not require a delimiting character. For example, the statement "GOTO100" is invalid, and must be expressed as "GOTO 100". However, in the statement "PRINT A;B;C", the semi-colon delimiters eliminate the need for trailing spaces in the variable list.

2,002 Line numbers

Each program line in a BASIC/Z program must begin with a valid line number. A line number must contain from 1 to 5 numeric characters. It must appear as a valid decimal number in the range of 0 through 65535, and may not contain embedded spaces. Line numbers indicate the order in which BASIC/Z program lines are stored, and may also be referenced by BASIC/Z statements such as GOTO, GOSUB, RESTORE, etc.

2,003 Character set

BASIC/Z recognizes all ascii characters, as listed in appendix B. Lower case characters are allowed, but, except in the case of string constants, they are internally converted to upper case when processed. For example, the variable 'APPLE' may also be referenced as 'apple'.

2,004 BASIC/Z data

BASIC/Z programs may utilize two data types, numeric and string. Numeric data may be classified as control, control/d, integer or real (i.e. floating point). String data may consist of any sequence of characters, including letters, digits, spaces, and special characters. A data item may be a constant or a variable.

2,005 Constants

A constant is any value which never changes throughout a program. It is expressed as its actual value. A constant may be either a string, or a numeric value.

2,006 String constants

A string constant is a sequence of up to 250 characters, enclosed in double quotes ("). Quotes within a string must be doubled (e.g. the constant A"B must be entered as "A""B"). The length of a string is the number of characters.

2,007 Numeric constants

A numeric constant may be an integer or a real number.

An integer is a positive or negative whole number. It may be expressed as a decimal number, or with radix format, in any number base from 2 through 36. An example of integer format:

Decimal integer:	-n...n	-12345	or	65535
Radix integer:	xxRr...r	16R7F	or	2R10110110

In the above examples, - is an optional sign, n is a decimal digit from 0 through 9, r is a number (0-9) or a letter (A-Z) which must be valid for the radix base specified, R indicates radix format, and xx is the number base. The range of a decimal integer is 1-5E (2*ISIZE-1) to 5E (2*ISIZE-1). The maximum value of an integer in radix format is 65535.

A real number is a positive or negative number which includes a decimal point, or a number expressed in scientific notation. The formats of a real number may be:

Real format:	-n...n.n...	-1.2345
Scientific format:	-n...n.nE-xx	12E-34

In the above examples, - is an optional sign for the number or the exponent, n is a number expressed using the digits 0 through 9 and a decimal point, E specifies scientific notation, and xx is the exponent expressed with the digits 0 through 9. The range of a real number may be from (-1E+63)+1 to (1E+63)-1.

2,008 Variables

Variables are names used to represent values which are utilized in a BASIC/Z program. Variables may be of type control, control/d, integer, real, or string. The amount of memory used for the storage of each variable may be defined with a SIZES statement. ISIZE defines the memory allocated for each integer, RSIZE for each real variable, and SSIZE the default for each string (although strings may be explicitly dimensioned to other than SSIZE). Control and control/d variables have a fixed size of one and two bytes respectively.

The value of any variable may be assigned explicitly in the program, or it may be computed as the result of one or more calculations. Before a value is assigned to a variable, it is assumed to be zero (0) for numerics, and null ("") for strings.

2,009 Variable names

BASIC/Z variable names may be any length, with all characters significant. It must begin with a letter, followed by any combination of letters, digits, and periods. If a name begins with FA or FN, it is assumed to be a call to a user-defined function (section 5). A variable name may not be a reserved word, but it may contain an embedded reserved word. For example, 'RUN' is not a valid variable name, but 'RUN.TIME' is acceptable.

2,010 Control variables

A control variable is designated by a valid variable name, immediately followed by an ampersand (&) as a type identifier. It is stored in one byte of memory, and has a range of from 0 to 255. The internal format is one binary byte. Examples of internal control variable format:

```
A& = 9          |09|
                ----

A& = 255        |FF|
                ----

A& = 256        Range error

A& = 31         |1F|
```

2.011 Control/d variables

A control/d variable is designated by a valid variable name, immediately followed by a pound sign (#) as a type identifier. As a "double-width" control variable, it is stored in two bytes of memory, and has a range of from 0 to 65535. The internal format is two binary bytes in low/high sequence. Examples of internal control/d variable format:

A# = 14	¦0E¦00¦
A# = -1	Range error
A# = 8618	¦AA¦21¦
A# = 65534	¦FE¦FF¦

2.012 Integer variables

An integer variable is designated by a valid variable name, followed by a percentum (%) as a type identifier. It is stored in ISIZE bytes of memory, and has a range of from 1-5E(2*ISIZE-1) to 5E(2*ISIZE-1). The default ISIZE is three (3), but it may be declared in the range of 3-9 with a SIZES statement. The internal format is two BCD digits per byte, stored in tens complement. Examples of internal integer variable format:

	ISIZE = 3	ISIZE = 5
A% = 1234	¦00¦12¦34¦	¦00¦00¦00¦12¦34¦
A% = 500000	Range error	¦00¦00¦50¦00¦00¦
A% = -1	¦99¦99¦99¦	¦99¦99¦99¦99¦99¦
A% = -5978	¦99¦40¦22¦	¦99¦99¦99¦40¦22¦

In this format, the number is positive if the most significant digit of the most significant byte is less than five (5), or negative if it is greater or equal to five (5).

A negative tens complement number may be converted to positive by subtracting it from 10^(ISIZE*2). A negative number may be converted to tens complement format by adding it to 10^(ISIZE*2).

2.013 Real variables

A real (i.e. floating point) variable is designated by a valid variable name, with no following identifier. It is stored in RSIZE bytes of memory, and has a range of from (-1E+63)+1 to (1E+63)-1. The precision is 2(RSIZE-1) decimal digits. The default RSIZE is five (5), but, depending upon the precision required, it may be declared in the range of 4-10 with a SIZES statement. The internal format is:

Byte 1: The most significant bit specifies the sign, and the remaining seven bits the exponent, in excess 64 notation.

Byte 2 through RSIZE: 2 BCD digits per byte.

Examples of internal real variable format:

	RSIZE = 4	RSIZE = 6
A = 1234	¦44¦12¦34¦00¦	¦44¦12¦34¦00¦00¦00¦
A = 123.456789	¦43¦12¦34¦56¦	¦43¦12¦34¦56¦78¦90¦
A = -123.456789	¦C3¦12¦34¦56¦	¦C3¦12¦34¦56¦78¦90¦
A = 123456789	¦49¦12¦34¦56¦	¦49¦12¦34¦56¦78¦90¦
A = .000123456789	¦3D¦12¦34¦56¦	¦3D¦12¦34¦56¦78¦90¦
A = 0	¦00¦00¦00¦00¦	¦00¦00¦00¦00¦00¦00¦

In the above examples, note that the least significant digits, in excess of the maximum implied by RSIZE, are truncated.

2.014 String variables

A string variable is designated by a valid variable name, immediately followed by a dollar sign ($) as a type identifier. Every string variable must have a maximum declared length, in the range of 1 through 250, and takes on a current length, in the range of 0 through 250, as values are assigned to it. If a string longer than the maximum length is assigned to a variable, any excess characters will be truncated.

BASIC/Z implies a default maximum string length (known as SSIZE) of forty (40) characters for any scalar (non-array) string variable. However, an optional SIZES statement may designate an explicit SSIZE, which takes precedence. Any scalar string may be explicitly declared to have a maximum length of other than SSIZE with a DIM or COMMON statement. Every string array must be explicitly declared with a DIM, DDIM, or COMMON statement, which includes a specific definition of the maximum length of each element of the array.

The internal format of a string variable is:

```
Byte 1   ^ maximum string length
Byte 2   ^ current string length
Byte 3-n ^ any character(s) - (n = max from byte 1 + 2)
```

Examples of internal string variable format, assuming a maximum length of 10:

```
A$ = "ABCDEF"  |0A|06|41|42|43|44|45|46|xx|xx|xx|xx|
               -----------------------------------

A$ = ""        |0A|00|xx|xx|xx|xx|xx|xx|xx|xx|xx|xx|
               -----------------------------------
```

In the above examples, xx specifies a byte which may take any value, as it has no significance.

2.015 Conversion of data types

BASIC/Z provides automatic conversion between all numeric data types to allow mixed-mode arithmetic. If a real value is converted to any other numeric type, the fractional part will be truncated, while preserving the original sign.

BASIC/Z provides numerous functions for conversion between numeric and string variables. They include, among others, ASC, CHR$, CVTC, CVTC$, FILL, FILLSPC, FMT, STR$, STRING$, and VAL.

2.016 Array variables

String or numeric data may be stored as an array. An array is a set or table of variables, which are referenced by the same variable name.

A numeric array is designated by a variable name, optionally followed by an ampersand, pound sign, or percentum (if it is other than real type). A numeric array may have from one (1) to four (4) dimensions. A string array is designated by a variable name, followed by a dollar sign. A string array may have from one (1) to three (3) dimensions.

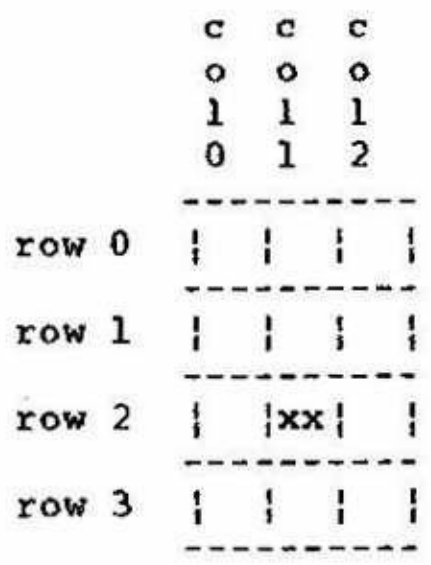
All arrays are zero indexed. Unlike some other dialects of BASIC, an implied array dimension of ten is not recognized by BASIC/Z. Every array must be explicitly declared with DIM, DDIM, or COMMON prior to any reference.

A one dimensional array, or vector, is a simple linear list in which the elements are stored sequentially in memory. For example, an array dimensioned to X(3) is stored:

X(0)
X(1)
X(2)
X(3)

An element of a one dimensional array is referenced by the array name, with the index enclosed in parenthesis, which may be a constant or an expression.

A two dimensional array is conceptualized as a table organized by rows and columns. An array dimensioned to A(3,2) would be represented as:



An element of a two dimensional array is referenced by the array name, with row and column indices enclosed in parenthesis, which may be constants or expressions. The element marked as xx above would be referenced as A(2,1), where the first index is the row, and the second is the column. Three and four dimensional arrays are extensions of this concept. An element of one of those arrays is referenced by the array name, with the appropriate number of indices enclosed in parenthesis.

continued - -

2,016 Array variables (continued)

The elements of a two dimensional array are stored sequentially in memory, in column major order. That is, an array dimensioned to A(3,2) is stored:

A(0,0)
A(1,0)
A(2,0)
A(3,0)
A(0,1)
A(1,1)
A(2,1)
A(3,1)
A(0,2)
A(1,2)
A(2,2)
A(3,2)

Likewise, the elements of a three dimensional array follow the same pattern. An array dimensioned to B(1,1,1) is stored:

B(0,0,0)
B(1,0,0)
B(0,1,0)
B(1,1,0)
B(0,0,1)
B(1,0,1)
B(0,1,1)
B(1,1,1)

While strings are limited to a maximum length of 250 characters, BASIC/Z offers a substitute data structure without this restriction, the control variable array. Each of these data structures reserve a contiguous series of memory locations, and store one character, or array element, in each byte of that block. BASIC/Z even provides three unique reserved words (FILL, FILLSPC, and STRING$) to insert and extract strings from control arrays. For example:

```
100 fill(x&(0),x&(4)) = "ABCDE"
110 y$ = string$(x&(1),x&(3))
```

```
100 mid$(x$,1) = "ABCDE"
110 y$ = mid$(x$,2,3)
```

The above two examples yield the identical result (y$ = "BCD") by two different methods. The advantage of the first is that the maximum size of the control array is limited only by available memory.

An extension of this concept might be applied to file handling by those comfortable with field statements in other dialects of BASIC. Used as a file buffer, the control array is dimensioned to the size of the logical record length, and GETVEC and PUTVEC statements are used to read and write the array to and from each record of the file. FILL and FILLSPC are used (in place of LSET from other dialects) to assign values into this file buffer, and STRING$ is used to extract them.

2,017 Numeric operators

Numeric operators specify arithmetic operations. They may be sub-classified as binary, which operate on two data items, and unary, which operate on a single data item.

The unary operators are:

-	Negation	-X
+	No effect	+1

The binary operators are:

^	Exponentiation	X ^ 2
/	Division	X / Y
*	Multiplication	X * Y
\	Integer Division	X \ Y
-	Subtraction	X - Y
+	Addition	X + Y

The "+" symbol is recognized as a unary operator to allow constructs such as X = +Y to be syntactically acceptable, even though the "+" has no effect.

2,018 String operators

Concatenation is the only operator allowed for string data items.

+	Concatenation	X$+Y$

The concatenation operator (+) will produce a string composed of the characters in the data item to the left of the operator, followed by the characters in the data item to the right of the operator. For example, if A$="CHIC" and B$="AGO" then A$+B$ yields the string value "CHICAGO".

2,019 Relational operators

Relational operators allow the comparison of the values of either numeric or string data items. The result returned by this comparison is a boolean value of either "true" (1) or "false" (0). The data items compared must both be of the same general type, either string or numeric.

=	Equal to	X = Y
<>	Not equal to	X <> Y
<	Less than	X < Y
>	Greater than	X > Y
<=	Less or equal	X <= Y
>=	Greater or equal	X >= Y

Expressions containing relational operators are most frequently found with DO/UNTIL, WHILE/WEND, or IF/THEN/ELSE statements as they require a boolean true/false condition to determine program flow. However, they may be used in many other ways, such as:

```
100  print  (x$=y$)
110  x = (y > 4.11)
```

In the above, if x$ and y$ are equal, a numeric value one (1) would be printed, while a numeric value zero (0) would be printed if they are not equal. Further, if y is greater than 4.11, x is set equal to one (1), while it is set to zero (0) if it is not greater. In the above examples, the relational expressions are used to return a numeric value of one or zero, rather than the actual concept of true/false boolean logic. In these cases, the expression must be enclosed in parenthesis, to avoid any possible ambiguity.

String comparisons are made by comparing the ascii codes, one character at a time, until there is a mis-match, or the end of a string is reached. If all codes are the same, the strings are equal. If there is a mis-match, the string containing the higher code is greater. If the end of one string is reached, the shorter string is considered smaller. Leading and/or trailing spaces are considered significant in this comparison.

2.020 Logical operators

BASIC/Z provides six logical operators to perform tests on multiple relations, bit manipulations, or boolean operations. The NOT operator is unary, and requires one operand, while the rest are binary, requiring two operands. Logical operators convert all operands to a 16 bit unsigned value (control/d variable format), and yield a 16 bit unsigned result. The outcome of a logical operation is determined as shown in the following table:

NOT: X	NOT X
1	0
0	1

XOR: X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

AND: X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

IMP: X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

OR: X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

EQV: X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Special care must be taken when assigning the result of a logical expression to a control variable:

```
100 x& = 0
110 x& = not x&
```

The above example would normally be used to just complement the value in x&, an acceptable operation. However, as logical operations are performed on 16 bit values, line 110 would generate a numeric range error upon the attempt to assign the value 16RFFFF to x&. The following should be used instead:

```
100 x& = 0
110 x& = not x& and 16RFF
```

2.021 Boolean values

In BASIC/Z, simple or complex expressions may be formed which evaluate to a single value of "true" (1) or "false" (0). This is referred to as a boolean value. They are used most frequently with DO/UNTIL, WHILE/WEND, or IF/THEN statements which require a true/false condition to determine program flow.

Truth or falsity is determined by converting a value to a 16 bit unsigned binary number. If the least significant bit is zero (0), the value is considered false. If it is one (1), the value is considered true. As all other bits in the number are ignored, this also provides an easy method of determining whether a number is odd or even:

```
100  if x# then gosub @print.it
```

In the above example, the subroutine @print.it is only executed if x# is an odd number. However, this method may only be used with values in the range of 0 through 65535.

2.022 Expressions

An expression may be a numeric or string constant, or a variable. Optionally, it may combine variables and/or constants with operators, to produce a single value.

Operators are symbols which specify mathematical or logical operations to be performed upon data items. The operators provided by BASIC/Z may be classified into four general groups: Numeric, String, Relational, and Logical.

2.023 Precedence in expression evaluation

BASIC/Z expressions are evaluated in the following sequence:

a) function references
b) relational string expressions
c) expressions in parenthesis
d) unary operators (+ - NOT)
e) exponentiation
f) multiplication / division
g) integer division
h) addition / subtraction
i) relational numeric expressions
j) AND
k) OR
l) XOR
m) IMP
n) EQV

2.024 Numeric output formats

A numeric data item is converted to string when it is output with a PRINT or PUT statement. Unless an output format is explicitly specified with a FMT function, it will take the following format:

1) The first position will be a minus sign or a blank space, depending upon the sign of the number.

2) If the number has more digits than that specified by RSIZE, it is output in scientific notation.

3) If the absolute value of the number is greater than 0 and less than 0.1, the number is output in scientific notation.

4) A trailing space is output as a delimiter.

3,000 Executive Commands

An executive command specifies an operation to be performed, from the command level of BASIC/Z. This allows BASIC/Z to be controlled from the system console.

Executive commands are entered by typing characters in sequence on the console keyboard. The command entry is terminated by depressing the <RETURN> key. During the entry of a command, each character typed is echoed to the console display. In command line entry, BASIC/Z will recognize the standard back-space and control-x (erase entire line) functions.

An executive command consists of the command word, optionally followed by one or more parameters. If a command line contains more than one parameter, each must be separated by a valid delimiter, a comma or a space. File name parameters may be enclosed in double quotes ("), but this is optional. Numeric parameters are expected in decimal, but radix format numbers are recognized for entry in another number base.

Any executive command may be entered by typing the full command-word. However, for many of the more frequently used commands, BASIC/Z will also recognize a convenient abbreviation. If an abbreviated version of the command is allowed, it will be noted on the 'Abbrev:' line of each page in this section.

Executive commands may not be inserted into your BASIC/Z programs. They are designed to be used interactively, from the command level of BASIC/Z only.

A detailed description of each available executive command follows in this section.

3.001 ATTACH

Format: ATTACH nlit

Abbrev: none allowed

Purpose: to attach a printer to this process

Example: ATTACH 1

In a multiple printer system, the ATTACH command is used to request that the printer specified by the numeric literal be attached to this process. The precise implementation of this command will depend upon the operating system in use.

When running under MP/M II, or equivalent, one of up to sixteen printers may be selected. Therefore, a valid argument to this command would be a numeric literal in the range of zero (0) through fifteen (15). In this environment, BASIC/Z will process your printer attach/detach requests through the XDOS functions 159/161.

When running under CP/M (Rev. #2.x), or equivalent, it is assumed that one of up to three printers may be selected. Therefore, a valid argument to this command would be a numeric literal in the range of zero (0) through two (2). In this environment, BASIC/Z will process your printer attach/detach requests through the iobyte, accessed by the BDOS functions 7/8. The two most significant bits of the iobyte are set as follows by attach/detach requests:

```
DETACH   -  00000000
ATTACH 0 -  01000000
ATTACH 1 -  10000000
ATTACH 2 -  11000000
```

When running under an operating system which does not support either of the above concepts, execution of the ATTACH command is disabled.

Certain operating systems (such as MP/M II and Vector Graphic Extended CP/M) will report that a requested printer is already in use by another process. In this instance, BASIC/Z will display the error message "Printer Busy".

see also: DETACH (section 3.013)

3.002 ATTRS

Format: ATTRS [dr:]filename nlit

Abbrev: none allowed

Purpose: to allow a change of file attributes

Example:
```
ATTRS  PROGRAM1  3
ATTRS  PROGRAM2  2R101
```

BASIC/Z recognizes three general file attributes: read/only, system, and archive. Attributes of a file are recorded in the high-order bits of the FCB file type (t1', t2', t3').

Read/only (t1'): write and erase operations to this file are not allowed.

System (t2'): the file name is normally not printed by a DISPLAY command or statement. Also, user zero system files may usually be accessed on a read only basis from other user numbers (although this depends upon the operating system in use).

Archive (t3'): designed for user written archive programs. Generally, the archive attribute is set when a file is copied to backup storage, and reset whenever the file is altered. This facility allows a program to determine whether a backup operation is needed. Some operating systems (such as MP/M II) reset the archive attribute of a file automatically upon writing to it.

File attributes may be altered from the command level by passing an attribute argument in the range of zero (0) to seven (7). The attribute value is encoded by reserving one bit position for each of the three attributes, with t1' in the least significant position.

attr	bits	description
0	0 0 0	[no attributes set]
1	0 0 1	read/only
2	0 1 0	system
3	0 1 1	read/only - system
4	1 0 0	archive
5	1 0 1	archive - read/only
6	1 1 0	archive - system
7	1 1 1	archive - system - read/only

The file name must be unambiguous (i.e. contain no ? nor *), and may be optionally enclosed in double quotes ("). The dr: is optional. If not present, the currently selected default drive will be assumed.

see also: ATTR (section 4.006) and ATTRS (section 4.007)

3.003 AUTO

Format: AUTO [nlit]

Abbrev: A

Purpose: to set the increment for auto line generation

Example: AUTO 20

To simplify program entry, BASIC/Z provides a means to systematically generate program line numbers. This feature may be invoked by typing a space as the first character of a line. Upon execution, BASIC/Z will retrieve the last line number used, add the "auto-increment", and display it on the console.

If the auto feature generates a line number that is already present in the program, an asterisk is displayed to warn that any input will replace the existing line. Entry of a control-x will cancel an auto-generated line number.

The AUTO command is utilized to establish the "auto-increment". If an increment parameter is entered, BASIC/Z will accept it as the new increment for the auto mode. If not, the increment will be reset to the default value of ten (10).

3.004 BATCH

Format: BATCH

Abbrev: none allowed

Purpose: to initiate compilation of a series of programs

Example: BATCH

Frequently, it is necessary to compile several programs in one session. This may become quite time consuming (and irritating when you must continually check if "this one is done"!). BATCH provides a solution, as it may be used to initiate automatic compilation of a large number of programs.

Upon execution of the BATCH command, BASIC/Z will display on the console:

'COMPILE '

At this time, just finish typing a compile command by adding a file name, optional drives, compiler options, and <RETURN>, just as you would with a single program compile. This sequence is repeated until you terminate the BATCH list by pressing just <RETURN> without a command line. BASIC/Z allows a maximum of 256 characters for the entire BATCH list, which will generally allow 20 or more entries. If it is exceeded, "Overflow - last entry deleted" will be printed, and command execution will proceed. After the final entry, BASIC/Z will inquire:

'Terminate batch on an error (y or n) ? '

If you answer yes, any compile-time error will cause BASIC/Z to abort the BATCH compile. Generally, this is the preferred option, as it avoids the possibility of erasing error messages from the console.

3.005 BIN

Format: BIN nlit

Abbrev: none allowed

Purpose: to display the binary representation of a number

Example: BIN 16R2C00

The command BIN will cause BASIC/Z to display a 16 bit binary representation of the specified numeric literal. The argument, if other than base 10 (decimal), must appear in radix format, in the range of 0 to 65535 (decimal).

3.006 BIND

Format: BIND

Abbrev: none allowed

Purpose: to create a command (.COM) file

Example: BIND

When a BASIC/Z program is compiled, an object file with an extension of ".BZO" is created. This is an executable, machine code form of the program. While it is "ready to run", it requires access to many general purpose subroutines (math, file handling, etc.) which are found in the RUN/Z run-time library.

For purposes of testing and debugging, a compiled program may be immediately executed by invoking RUN/Z as an executable program. This eliminates the drudgery of "linking" procedures after each modification. However, once a final version of your program has been compiled, the BIND command may be used to create an executable command (.COM) file, by binding together your object program with support routines from the run-time library. In this format, compiled programs are easier to use (they execute by typing your program name, not RZ xxxx, they allow single file disk transfers, etc.). Also, as the BASIC/Z license agreement does not allow distribution of RUN/Z as a stand alone program, you must BIND at least one program for packages that are distributed.

Generally speaking, you should BIND only one program out of any series of compiled programs, as all run-time routines will be included. Since these support routines are present in memory, your program can "RUN" or "CHAIN" a compiled object program (type .BZO) without wasting disk space or increasing load times as in some other compilers.

Upon execution of the BIND command, you will be asked to enter the names of the run-time library, the compiled object program, and the destination command file. Each may include an optional drive name, if other than the currently selected default drive. Since the extention of each of these files is pre-determined (COM/BZO/COM), that information is not needed in your response. As the new file is constructed, BASIC/Z will continuously display the current record number. This may help you gauge the approximate time to completion. The revision number (both major and minor part) of the compiler, run-time library, and compiled object program must be identical in order to execute a BIND command successfully.

The INSTALL utility may be used to configure either RUN/Z directly, or a command file created with BIND. However, if you regularly create programs for more than one system, you may find it more convenient to pre-configure several copies of RUN/Z (naming them RZ-DEC.COM, RZ-IBM.COM, RZ-LEAR.COM, etc.), and just choose the appropriate one when you execute BIND.

3.007 CHANGE

Format: CHANGE

Abbrev: C

Purpose: to locate all occurrences of a series of characters, and conditionally replace each with a second series

Example: CHANGE

BASIC/Z will prompt: 'Search mask ? '

A string of up to 132 characters may be entered, followed by <RETURN>. If no characters are entered, BASIC/Z will cancel the command. "Wild-Card" substitution is recognized in the search mask. Each question mark (?) that appears is treated as a match for any character in that position. For example, the mask A?C will match on ASC, A(C, etc.

BASIC/Z will prompt: 'Change to ? '

A string of up to 132 characters may be entered, followed by <RETURN>. BASIC/Z will search through the program buffer to locate the first occurrence of the specified mask. If an exact match is not found, the message 'String not found' will be displayed.

Due to the syntax of the BASIC/Z language, it is necessary that any replacement be made on a conditional basis. For example, an attempt to change the real variable 'A' would also cause a change to A%, A$, A1, A(X), etc. Therefore, each time a match is found, BASIC/Z will display it in the following format:

```
100 LET B(X) = A+B1+C1
100 LET B(X) = ^  Y or N ?
```

Entry of a "Y" will cause BASIC/Z to replace the search mask string with the replacement string in the program line. Entry of an "N" (or any other character) will cause BASIC/Z to disregard it entirely.

If replacement would cause a syntax error, the line will be left unchanged, with an appropriate error message displayed on the console.

3.008 CLEAR

Format: CLEAR

Abbrev: CLR

Purpose: to delete program data in memory

Example: CLEAR

Execution of the command CLEAR will cause BASIC/Z to delete all program data stored in memory. This includes the source program buffer, the title buffer, and the counter for the next line number to be "auto-generated".

However, all other data (such as the "auto-increment", pagesize, etc.) remains unaffected.

3.009 CLS

Format: CLS

Abbrev: none allowed

Purpose: to erase the console screen

Example: CLS

Upon execution of CLS, BASIC/Z will erase the console screen and position the cursor to "home" (upper left corner of the screen).

3.010 COMPILE

Format: COMPILE [filename] [dr:] [dr:] [&options]

Abbrev: none allowed

Purpose: to compile a BASIC/Z program

Example: COMPILE TEST B: A: &POI(C:)

The COMPILE command causes BASIC/Z to compile the specified source program (type .BZS) to an object file (type .BZO) on the first dr:, using the second dr: for work space. If filename is not included, BASIC/Z will default to the name specified by the TITLE command. If dr: names are not specified, the default will be assumed. During the compilation process, BASIC/Z will display a "count" from 0 through 5, to help gauge the time to completion.

A compiler option is chosen by including the key-letter, from the selection below, in the option parameter. If present, it must be the last parameter in the command line. If the argument filename is not provided, this string must begin with an ampersand (&) to distinguish it from a program name.

OPTIONS :

C - CONSOLE ~ Output will be displayed on the system console.

D - DEBUG ~ BASIC/Z will recognize any DEBUG statements which are present in the source file. Otherwise, DEBUG statements are ignored.

E - ECHO ~ Output will be echoed to both the console and printer.

F - FILE ~ Compiler output will be directed to a disk file (with an extension of .PRN) on the default drive, or a specified drive by including the name enclosed in parenthesis, such as: &F(C:).

I - INCLUDE DRIVE ~ The default drive for files to be included is specified by the name enclosed in parenthesis, such as: &I(A:).

L - LISTING ~ The listing only option will suppress the creation of an object program.

N - NULL ~ All output to the print stream, other than error messages, will be suppressed. This is the default condition.

O - OBJECT ~ Assembly language source code for the compiled object program will be be included with the listing.

P - PRINTER ~ Output will be directed to the printer.

T - TABLE ~ A symbol table of program variables will be output to the print stream. If options C, E, and F are not present, then the table will be output to the system printer, without printing the program source code.

3.011 DEC

Format: DEC nlit

Abbrev: none allowed

Purpose: to display the decimal representation of a number

Example: DEC 2R10101111

The command DEC will cause BASIC/Z to display the decimal representation of the specified numeric literal. The argument, if other than base 10 (decimal), must appear in radix format, in the range of 0 to 65535 (decimal).

3.012 DELETE

Format: DELETE lnum [lnum]

Abbrev: DELT or DLT

Purpose: to delete one or more program lines

Example: DELETE 100 200

If the command is followed by a single line number, then only that line will be deleted. BASIC/Z will also accept the implied form of this command. Enter just the line number, followed by pressing <RETURN>, to delete one line.

If the command is followed by two line numbers, then program lines will be deleted starting at the first line number (or the next higher number that exists), through the second line number (or the next lower number that exists).

3.013 DETACH

Format: DETACH

Abbrev: none allowed

Purpose: to release the attached printer to other users

Example: DETACH

In a multiple printer system, DETACH is used to release the printer currently attached to this process. The precise implementation of this command will depend upon the operating system in use. Specific information on the methods used will be found under the ATTACH command in section 3.001.

When running under an operating system which does not support multiple printers, execution of the DETACH command will be disabled.

3.014 DISPLAY

Format: DISPLAY [dr:] [filename]

Abbrev: DIR or D

Purpose: to display a disk directory

Example: DISPLAY B:*.BZ?

The command DISPLAY will cause BASIC/Z to display, on the console, a formatted listing of all files on a specified drive which match the designated mask string. The mask string, if present, may be optionally enclosed in double quotes ("). A question mark (?) will match any character, and an asterisk (*) will match any name or type. If a mask is not included, BASIC/Z will assume the ambiguous mask "*.*", and display all files. If a dr: is not included, BASIC/Z will assume the currently selected default drive.

Immediately following the directory listing, the amount of unallocated disk space (in kilobytes) will be displayed.

3.015 DISPLAYP

Format: DISPLAYP [dr:] [filename]

Abbrev: DIRP or DP

Purpose: to print a disk directory on the printer

Example: DISPLAYP C:

The command DISPLAYP is identical to the command DISPLAY, with the single exception that all output is directed to the system printer.

3.016 DOS

Format: DOS

Abbrev: none allowed

Purpose: to transfer control to the disk operating system

Example: DOS

The command DOS will cause BASIC/Z to immediately transfer control to the disk operating system.

3.017 EDIT

Format: EDIT lnum

Abbrev: E

Purpose: to edit a specified program line

Example: EDIT 100

The command EDIT will allow you to change any portion of a specified program line without re-typing the entire line. Even the line number may be changed, with optional retention of the original line. However, if a line is changed such that it contains a syntax error, BASIC/Z will force you to re-edit the line until the error is corrected. Of course, the EDIT mode may be aborted at any time with the special edit command Q (quit), but this will cause the program line under edit to be discarded.

If the specified lnum appears in the program, the entire line will be displayed with the cursor positioned under the first digit of the line number. The cursor is always positioned below the original text, to avoid obscuring data from the operator's view.

BASIC/Z provides a set of special edit commands to facilitate rapid and simple changes to the program line. In order to maintain correct cursor positioning, the special edit commands are not echoed back to the console.

Most of the special edit commands may be automatically repeated by preceding the command with a decimal number in the range of 2 through 250. For example, 2SR will cause the cursor to search to the second occurrence of 'R' in the program line.

A detailed definition of each special edit command follows in this section:

Advance {SPACE BAR} Each time that the space bar is pressed, the cursor will advance to point to the next character in the program line. This may be done manually, or in the repeat mode, by preceding the space with a decimal number in the range of 2 through 250. The cursor will not move past the end-of-line position, unless you enter the append, insert, or replace mode.

Backspace {BACK-SPACE} Each time one of the three pre-defined back-space keys are pressed, the cursor will be moved one position to the left. In the normal edit mode, no change is made to the line. However, in the insert mode, each character passed over is deleted from the line.

continued - -

3.017 EDIT (continued)

Tab {TAB} Each time that the TAB (control-i) key is pressed, the cursor will be advanced to the next column of eight positions each.

Abort {ESC} Several special edit comands require, or allow, multiple codes for completion. In case of an entry error, a partial command may be aborted by pressing the escape key. No change is made to the edited line, but BASIC/Z remains in the edit mode to allow entry of additional special edit commands.

Append {A} The cursor is advanced to the end-of-line position, and the insert mode is entered to append additional text. The edit mode may be completed with a carriage return, and the changed line will be restored to the program buffer. You may exit the insert mode with the escape key, allowing for entry of additional special edit commands.

Change {C} A character may be changed by typing a C followed by the correct new character. The new character will be displayed, and the cursor will be advanced to the next position. Multiple characters may be changed, as a group, by preceding the command with a decimal number in the range of 2 through 250.

Delete {D} A character may be deleted by entering the command D. The deleted character will be displayed, enclosed in backslashes (\), and the cursor will be advanced to the next position. Multiple characters may be deleted, as a group, by preceding the command with a decimal number in the range of 2 through 250.

End Edit {E} or {RETURN} The edit mode is ended with the command E or with a carriage return. The edited line replaces the old line in the program buffer. If, during the edit process, the line number was changed, the edited line will be inserted at the correct new location in the program buffer, and the following message will be displayed:

'Line # was changed - Delete the old line ? '

A response of y will delete the old line entirely, while a response of n will leave the old line in the program buffer, in its original unedited format.

Insert {I} Characters may be inserted at the cursor location, by typing the command I, followed by the character(s) to be inserted. The insert mode can be exited by pressing the escape key, to remain in the edit mode. The edit mode may be ended with a carriage return, which will cause the new, edited version of the line to replace the old line in the program buffer. In the insert mode, back-space is operative, but every character passed over is deleted.

Jump {J} The cursor will "jump" over the line number and be positioned at the first statement of the program line.

continued - -

3.017 EDIT (continued)

Kill {K} Characters may be deleted from the cursor location up to, but not including, a specified search character by entering the command K, followed by the search character. Deleted characters will be displayed, enclosed in backslashes (\), and the cursor will advance to the next position. BASIC/Z will kill to the nth occurrence of a specified character, by preceding the command K with the number 'n'.

List {L} The entire line will be displayed, in its present, edited format. The cursor will then be positioned at the first character of the line. This is the only command which will re-position the cursor to the beginning of the line, without deleting text.

Move {M} The cursor will move to the position in the line of the most recently detected syntax error.

Quit {Q} The edit mode may be aborted entirely with this command. The program buffer will remain unchanged, and the edited version of the line will be lost.

Replace {R} Characters may be replaced by typing R, followed by the correct characters. If the cursor reaches the end-of-line position, the insert mode is automatically entered, thus appending each additional character to the end of the program line. When all characters have been replaced, press the escape key to exit the replace mode, while remaining in the edit mode. Optionally, press <RETURN> to end the edit mode, and replace the edited line in the program buffer.

Search {S} The cursor will be advanced to the first occurrence of a specified character by entering the command S, followed by the search character. BASIC/Z will search to the nth occurrence of a specified character, by preceding the command S with the number 'n'. For example, the command 2SR will search to the second occurrence of R. If the character is not found, the cursor will stop at the end-of-line position.

Zap {Z} The remainder of the program line, starting at the cursor position, is deleted as a group. The deleted character(s) will be displayed, enclosed in backslashes (\).

3.018 EXEC

Format: EXEC [filename] [&command]

Abbrev: RUN/Z or RZ

Purpose: to execute a BASIC/Z object program

Example: EXEC B:PROGRAM2 &PASSWORD

The command EXEC will cause BASIC/Z to load RUN/Z, followed by load and execution of the specified object program (type .BZO). If the argument filename is not included, BASIC/Z will default to the current program name specified by the TITLE command.

The &command parameter is optional. If present, it may be retrieved with the function COMMAND$. This feature could allow you to "password-protect" a BASIC/Z program. If the argument filename is not included, the &command must begin with an ampersand (&) to distinguish it from a program name (otherwise, the ampersand is optional). In any case, a leading ampersand in &command is never passed to the function COMMAND$, but treated only as a delimiter.

3.019 FILE

Format: FILE

Abbrev: F

Purpose: to display current program information

Example: FILE

The command FILE will cause BASIC/Z to display the program title, the lowest line number in the program, the highest line number in the program, and the total size of the source program, as a decimal number, in bytes.

3.020 FORMFEED

Format: FORMFEED

Abbrev: FF

Purpose: to set the printer to "top-of-form"

Example: FORMFEED

The command FORMFEED will cause the printer to be advanced to the "top-of-form" position. This is accomplished by outputting either a form-feed (0CH), or through consecutive line feeds, depending upon the form-feed status as set at the time of installation.

3.021 FREE

Format: FREE [dr:]

Abbrev: none allowed

Purpose: to display the amount of unallocated disk space

Example: FREE C:

Upon execution of the command FREE, BASIC/Z will display the number of kilobytes of available free space on the specified drive. If the argument dr: is not included, the current selected default drive will be assumed.

3.022 HEADER

Format: HEADER [any ascii characters]

Abbrev: H

Purpose: to provide a header message on listings and compilations

Example: HEADER Listing date - 6/1/83

The HEADER string, if defined, will be printed at the top of each page of program listings and compilations. At initialization, the HEADER string will be set to null so that nothing will be printed.

If only the command word is typed, the current HEADER string will be displayed on the console.

If the command word is followed by a text argument, the HEADER string will be set equal to the specified text. The new HEADER string may be optionally enclosed in double quotes ("). BASIC/Z will acknowledge the new HEADER string by displaying it on the console.

The maximum length of a HEADER string is forty characters. Any text in excess of this length will be disregarded. The HEADER option is particularly useful for documenting the date and/or time of a listing or compilation.

3.023 HEX

Format: HEX nlit

Abbrev: none allowed

Purpose: to display the hexadecimal representation of a number

Example: HEX 8R377

The command HEX will cause BASIC/Z to display a four digit hexadecimal representation of the specified numeric literal. The argument, if other than base 10 (decimal), must appear in radix format, in the range of 0 to 65535.

3.024 LINK

Format: LINK filename [&command]

Abbrev: none allowed

Purpose: to execute a command file (type .COM)

Example: LINK PIP C:=A:*.*[V]

The LINK command will allow you to load and execute a command file. Upon execution, the command file named by the filename argument is loaded into the TPA (transient program area), and executed at the TBASE address. As the command file is assumed to have an extension of ".COM", it need not be included in the argument. The filename must be unambiguous (i.e. contain no ? nor *), and may be optionally enclosed in double quotes ("). If a drive name is not included, the currently selected default drive will be assumed.

The &command parameter is optional. If present, it will be passed to the command file as a command line trailer. The argument &command may begin with an optional ampersand, to distinguish it from a filename argument. However, a leading ampersand is never passed to the command file, but treated only as a delimiter.

3.025 LIST

Format: LIST [lnum] [lnum]

Abbrev: L

Purpose: to display all or part of a program on the console

Example: LIST 100 200

If only the command word is typed, the entire program will be displayed.

If the command word is followed by a single line number, the program will be displayed starting at the specified line number (or the next higher number that exists), through the last line of the program.

If the command word is followed by two line numbers, the program will be displayed starting at the first line number (or the next higher number that exists), through the second line number (or the next lower number that exists).

All output from this command is handled by our "interruptable speed-control". For a full description of this feature, please refer to section 1.011.

3.026 LISTP

Format: LISTP [lnum] [lnum]

Abbrev: LP

Purpose: to list all or part of a program on the system printer

Example: LISTP 100 200

The command LISTP is functionally identical to the command LIST, with the exception that all output is directed to the system printer.

Listings generated by this command will be properly paginated to the PAGESIZE parameters specified. Each page will be numbered, and also include the title, as well as a HEADER line, if one was designated.

3.027 LOAD

Format: LOAD [filename] [&L]

Abbrev: LD

Purpose: to load a BASIC/Z source program into the program buffer

Example: LOAD B:PROGRAM1

The command LOAD will cause BASIC/Z to load a source program into the program buffer. If the filename argument is entered, it may be optionally enclosed in double quotes ("). If a drive reference is not included, the currently selected default drive will be assumed. If the filename argument is not entered, BASIC/Z will default to the current program name specified by the TITLE command (particularly convenient for re-loading after a compile).

An implied form of the LOAD command is also recognized. At the time of execution of BASIC/Z, the name of a source program to be loaded may be included as a command line trailer. For example, from the operating system level, a command line of: 'BZ PROGRAM1 ' would cause BASIC/Z to be executed, followed by a LOAD of the source program 'PROGRAM1.BZS '. Similarly, during program testing and debugging, the final executable statement in your program might be: LINK "BZ", "PROGRAM1" to achieve the same result.

If the specified program name does not include a type, BASIC/Z will assume it is a standard source file (type .BZS). It must have been saved to disk with a version of BASIC/Z bearing the same major revision number, and the same or earlier minor revision number. If this criteria is not met, the command will be aborted, and the error message "Wrong file type" will be displayed.

If the specified program name has a type other than .BZS, it will be assumed that the file is in ascii format. An ascii file with a type consisting of blank spaces may be loaded by including a trailing period (.) in the filename argument (e.g. LOAD PROGRAM1.). As each line of an ascii file is loaded, it is tested for proper syntax. If a valid line number is not present, the entire line must be discarded (unless the &L option is invoked). If any syntax errors are found, the line will be converted to a remark by inserting a single quote (') at the first position of the line. Any such errors may then be easily located with SEARCH, and corrected with EDIT.

If the option &L is included at the end of the command line, BASIC/Z will insert a line number at the beginning of each line, as it is loaded. This option is useful for loading ascii format program files which were created with an editor other than BASIC/Z.

3.028 LOWCASE

Format: LOWCASE

Abbrev: LC

Purpose: to allow lower case input from the console

Example: LOWCASE

BASIC/Z provides a software "cap-lock" option at the command level. If enabled (with the command UPCASE - section 3.043), all alphabetic console input is forced to upper case as it is typed. Execution of the command LOWCASE allows lower case input from the system console.

3.029 MERGE

Format: MERGE filename

Abbrev: none allowed

Purpose: to merge a BASIC/Z source program with the program in memory

Example: MERGE B:PROGRAM1.ASC

The MERGE command appends, inserts, or replaces program lines in the memory resident program, based upon the comparative line numbers. In the case of duplicate line numbers, the file merging from disk always takes precedence by replacing the line stored in memory.

This command will allow you to maintain a library of subroutines, which can be utilized immediately, without extensive re-typing.

The file to be merged from disk must have been saved in ascii format, with a type other than .BZS. The filename argument may be optionally enclosed in double quotes ("). If a drive reference is not included, the currently selected default drive will be assumed. An ascii file with a type consisting of blank spaces may be merged by including a trailing period (.) in the filename argument (e.g. MERGE PROGRAM1.).

As each line of the ascii file is merged, it is tested for proper syntax. If a valid line number is not present, the entire line must be discarded. If any syntax errors are found, the line will be converted to a remark by inserting a single quote (') at the first position of the line. Any such errors may then be easily located with SEARCH, and corrected with EDIT.

3.030 OCT

Format: OCT nlit

Abbrev: none allowed

Purpose: to display the split octal representation of a number

Example: OCT 16R1FFF

The command OCT will cause BASIC/Z to display a six digit split octal representation of the specified numeric literal. The argument, if other than base 10 (decimal), must appear in radix format, in the range of 0 to 65535.

3.031 PAGESIZE

Format: PAGESIZE [lines] [columns]

Abbrev: P

Purpose: to allow user control of printer pagesize

Example: PAGESIZE 66 132

The command PAGESIZE allows user control of the system printer form size to assure that all listings will be properly paginated. If a PAGESIZE command is not executed, BASIC/Z will default to the sizes entered at installation.

If only the command is entered, the current parameters will be displayed on the console.

If the command is entered with a single parameter, BASIC/Z will change the lines per page value only, and display both parameters.

If the command is entered with two parameters, BASIC/Z will change both values and display both parameters.

3.032 PRINTER

Format: PRINTER nlit

Abbrev: PR

Purpose: to output a user-defined series of control codes to the printer

Example: PRINTER 6

The PRINTER command is designed to allow easy control of font size or other printer logic directly from the system console. Upon execution, BASIC/Z will output a user-defined set of control codes to the system printer.

Up to eight (8) sets of control codes may be defined with the INSTALL utility program. Depending upon the value of the argument nlit (which must range between 0 and 7), the corresponding set of control codes is output.

3.033 RENAME

Format: RENAME

Abbrev: REN

Purpose: to rename a disk file

Example: RENAME

The RENAME command is used to change the name of an existing disk file. Upon execution of the command:

BASIC/Z will prompt: 'Old file name: '

Type the current name of an existing disk file which you wish to alter, followed by pressing <RETURN>. If an optional drive name is not included, the currently selected default drive is assumed. If the file cannot be located, the RENAME command will be aborted.

BASIC/Z will prompt: 'New file name: '

Type the new name you desire for this file, followed by pressing <RETURN>. If the new name includes a drive reference, it will be ignored, as the drive is established by the first response.

3.034 RENUM

Format: RENUM [option parameters]

Abbrev: none allowed

Purpose: to systematically renumber program lines

Example: RENUM 100 900 5

The entire program may be renumbered, or optionally, just a portion of it. All references to a changed line number are updated, including those appearing after a GOTO, GOSUB, RESTORE, etc.

There are four general forms to the RENUM command. In the following examples, I=Increment, F=First line to change, and N=New line number for the first line changed.

Enter RENUM and BASIC/Z will renumber the entire program. The first line in the program will be renumbered to 10, and each subsequent line will be incremented by 10.

Enter RENUM I and BASIC/Z will renumber the entire program. The first line in the program will be renumbered to I, and each subsequent line will be incremented by I.

Enter RENUM F I and BASIC/Z will renumber lines starting at line number F. Line number F will remain unchanged, and each subsequent line will be incremented by I.

Enter RENUM F N I and BASIC/Z will renumber lines starting at line number F. Line number F will be changed to line number N, and each subsequent line will be incremented by I. RENUM may not be used to alter the sequence of program lines. Therefore, in this form of the command, N must always be equal to, or greater than F.

3.035 RESAVE

Format: RESAVE [dr:] [&A]

Abbrev: none allowed

Purpose: to save the current source program, writing over an existing disk file with the same file name

Example: RESAVE C:

Upon execution, the current BASIC/Z source program in the program buffer is written to disk to replace a previous file by the same name. If a previous file is not found, BASIC/Z will display the error message "File not found", and abort the command. In this instance, the SAVE command should be used instead.

The dr: is optional. If not specified, BASIC/Z will default to the drive from which the file was originally loaded.

BASIC/Z must know the name of a source program before a SAVE or RESAVE is executed. This is automatic if the file was loaded, or may be manually entered or changed with the TITLE command.

If the option parameter &A is included in the command line, the program will be saved in ascii format. If the file was originally loaded as an ascii file, then the previous file type will be used. If a file type was specified with a TITLE command, then that type will be used. In any other case, ascii format files will be assigned a type of blank spaces.

3.036 RESET

Format: RESET [nlit]

Abbrev: none allowed

Purpose: to reinitialize the operating system after disk changes

Example: RESET 2R101

When a disk is changed, the operating system considers it read/only until a RESET is executed. RESET must not be executed until the affected disk drives are ready. If the nlit is present, it is used as a bit pattern to select which drives to reset. The least significant bit corresponds to drive A, through the sixteenth bit, which corresponds to drive P. For example, an argument of 5 (equivalent to 2R101) would reset only drives A and C.

3.037 SAVE

Format: SAVE [dr:] [&A]

Abbrev: none allowed

Purpose: to save the current source program to disk

Example: SAVE B:

Upon execution, the current BASIC/Z source program in the program buffer is written to disk. If a previous file of the same name is found, the error message "Duplicate file name" will be displayed, and the command aborted. In this instance, the RESAVE command should be used instead.

The dr: is optional. If not specified, the currently selected default drive will be assumed.

BASIC/Z must know the name of a source program before a SAVE or RESAVE is executed. This is automatic if the file was loaded, or may be manually entered or changed with the TITLE command.

If the option parameter &A is included in the command line, the program will be saved in ascii format. If the file was originally loaded as an ascii file, then the previous file type will be used. If a file type was specified with a TITLE command, then that type will be used. In any other case, ascii format files will be assigned a type of blank spaces.

3.038 SCRATCH

Format: SCRATCH filename

Abbrev: ERA

Purpose: to erase one or more files on disk

Example: SCRATCH A:FILE2.DAT

The command SCRATCH will cause BASIC/Z to erase one or more files from the directory of the specified disk. The filename argument may be optionally enclosed in double quotes ("), and may contain ambiguous references. A question mark (?) will match any character, and an asterisk (*) will match any name or type. If a drive reference is not included, the currently selected default drive will be assumed.

3.039 SEARCH

Format: SEARCH

Abbrev: S

Purpose: to locate all occurrences of a series of characters

Example: SEARCH

BASIC/Z will prompt: 'Search mask ? '

A string of up to 132 characters may be entered, followed by <RETURN>. If no characters are entered, BASIC/Z will cancel the command. "Wild-Card" substitution is recog-nized in the search mask. Each question mark (?) that appears is treated as a match for any character in that position. For example, the mask A?C will match on ASC, A(C, etc.

BASIC/Z will now search through the program buffer to locate every occurrence of the specified search mask. If an exact match is not found, the message "String not found" will be displayed.

Each time that an exact match is located, BASIC/Z will display the entire program line on the console. All output from this command is handled by our "interruptable speed-control", as defined in section 1.011.

3.040 SEARCHP

Format: SEARCHP

Abbrev: SP

Purpose: to locate all occurrences of a series of characters, listing them on the printer

Example: SEARCHP

The command SEARCHP is functionally identical to the command SEARCH, with the exception that all output is directed to the printer.

3.041 SELECT

Format: SELECT [dr:]

Abbrev: SEL

Purpose: to set or display the current default drive

Example: SELECT A:

The command SELECT allows control of the current default drive from the command level of BASIC/Z.

If only the command is entered, BASIC/Z will display the name of the current default drive.

If the command is entered with a drive name, the current default drive will be changed to the drive specified.

3.042 TITLE

Format: TITLE [filename]

Abbrev: T

Purpose: to create, change, or retrieve the program title

Example: TITLE NEWNAME.ASC

BASIC/Z must know the name of a source program before a SAVE or RESAVE is executed. This is automatic if the file was loaded, or may be manually entered or changed with the TITLE command.

If only the command is entered, BASIC/Z will display the current title on the console. If the title includes a type designation for ascii format, it will also be displayed, enclosed in brackets ([]).

If a new title is entered, it may be optionally enclosed in double quotes ("), and may also include a file type designation other than ".BZS". If a type is included, it will automatically be used for any SAVE or RESAVE of the program in ascii format.

3.043 UPCASE

Format: UPCASE

Abbrev: UC

Purpose: to force alphabetic console input to upper case

Example: UPCASE

BASIC/Z provides a software "cap-lock" option at the command level. If enabled (with the command UPCASE), all alphabetic console input is forced to upper case as it is typed. Execution of the command LOWCASE (section 3.028) allows lower case input from the system console.

3.044 USER

Format: USER [nlit]

Abbrev: U

Purpose: to change or retrieve the current user number

Example: USER 7

The USER command will allow you to display or to alter the current user number.

If only the command is entered, BASIC/Z will display the current user number. If the command is followed by a numeric literal, the current user number will be changed to that which was typed. Depending upon the operating system in use, a valid user number may range from 0-15, or from 0-31.

3.045 VIDEO

Format: VIDEO nlit

Abbrev: V

Purpose: to output a user-defined series of control codes to the console

Example: VIDEO 4

The VIDEO command is designed to allow easy control of various user-defined console attributes. Upon execution, BASIC/Z will output a pre-defined set of control codes to the console.

Up to eight (8) sets of control codes may be defined with the INSTALL utility program. Depending upon the value of the argument nlit (which must range between 0 and 7), the corresponding set of control codes is output.

3.046 XRF

Format: XRF

Abbrev: none allowed

Purpose: to execute the optional cross-reference generator

Example: XRF

Upon execution, the optional cross-reference generator (XRF.COM) will be loaded from disk and executed.

4,000 Reserved words

BASIC/Z statements specify operations to be performed in a program, and describe the data elements involved in the program flow. Each consists of a reserved word, optionally followed by a list of expressions which designate the operation(s) required.

Functions are included in BASIC/Z to provide commonly required computations. A function reference consists of the function name, which may be followed by the function arguments, if required. Arguments, if present, must be enclosed in parenthesis, and each must be delimited from the next by a comma. The opening parenthesis must immediately follow the function name, with no intervening blank spaces. A function always returns a single value, which may be of numeric or string type, depending upon the function definition.

BASIC/Z recognizes two general types of functions: intrinsic functions and user-defined functions. User-defined functions are fully described in section 5 of this reference manual.

Intrinsic functions are pre-defined as a part of BASIC/Z. They may be further sub-classified as either numeric or string, depending upon the type of data they return to the calling expression.

Numeric functions provide most of the commonly used mathematical functions. They always return data of numeric type. These functions are computed with up to eighteen digits of precision.

String functions are provided to manipulate and compare strings and sub-strings. They always return data of string type.

A detailed description of each available reserved word follows in this section.

4.001 ABS

Format: ABS(x)

Type: intrinsic function

Example: A = ABS(2*X)

The ABS function returns the absolute numeric value of the expression x. If x is greater or equal to 0, it is returned unchanged. If x is less than 0, the negated value of x is returned.

4.002 ALTKEY

Format: ALTKEY

Type: intrinsic function

Example: IF ALTKEY THEN GOTO @SCREEN.HANDLER

The function ALTKEY is used to determine which key was pressed to terminate the most recently executed INPUT$ or EDIT$ function. This value can then be used to determine future program flow.

ALTKEY returns a value of numeric type. If the <RETURN> key was pressed, it returns a logical false or zero (0) result. If one of the ten alternate terminator keys was pressed, it returns a logical true result, which could be 1,3,5,7,9,11,13,15,17,19 corresponding to the key which was pressed.

4.003 ASC

Format: ASC(x$)

Type: intrinsic function

Example: A = ASC(X$)

The function ASC returns a numeric value equal to the ascii code of the first character of the expression x$. If x$ evaluates to a null string, the value zero (0) is returned.

4.004 ATN

Format: ATN(x)

Type: intrinsic function

Example: A = ATN(X)

The ATN function returns a numeric value equal to the arctangent of the expression x. The value returned will be in the range of -pi/2 to pi/2.

The argument x must be expressed in radians. Degrees may be converted to radians by dividing the number of degrees by 2pi, or with the function RAD, described in section 4.160.

4.005 ATTACH

Format: ATTACH(x)

Type: intrinsic function

Example: IF NOT ATTACH(1) THEN GOTO @PRINTER.BUSY.ROUTINE

In a multiple printer system, the ATTACH function is used to request that the printer specified by the numeric expressiion x be attached to this process. The precise implementation of this function will depend upon the operating system in use.

Certain operating systems (such as MP/M II and Vector Graphic Extended CP/M) will report whether a printer ATTACH request was completed. In this instance, the ATTACH function will return a numeric value of true (1) if the request was successful, or false (0) if not. In all other cases, as success must be presumed, a result of true (1) is returned.

When running under MP/M II, or equivalent, one of up to sixteen printers may be selected. Therefore, a valid argument to this function would be a numeric expression which evaluates in the range of zero (0) through fifteen (15). In this environment, BASIC/Z will process your printer attach/detach requests through the XDOS functions 159/161.

When running under CP/M (Rev. #2.x), or equivalent, it is assumed that one of up to three printers may be selected. Therefore, a valid argument to this function would be a numeric expression which evaluates in the range of zero (0) through two (2). In this environment, your attach/detach requests will be processed through the iobyte, accessed by the BDOS functions 7/8. The two most significant bits of the iobyte are set as follows:

```
DETACH   -  00000000
ATTACH 0 -  01000000
ATTACH 1 -  10000000
ATTACH 2 -  11000000
```

see also: DETACH (section 4.042)

4.006 ATTR

Format: ATTR(x)

Type: intrinsic function

Example: A = ATTR(1)

The ATTR function is used to determine the current attributes of a disk file. It returns a numeric value in the range of zero (0) through (7), indicating the attributes of the file associated with file #x.

BASIC/Z recognizes three general file attributes: read/only, system, and archive. Attributes of a file are recorded in the high-order bits of the FCB file type (t1', t2', t3').

Read/only (t1'): write and erase operations to this file are not allowed.

System (t2'): the file name is normally not printed by a DISPLAY command or statement. Also, user zero system files may usually be accessed on a read only basis from other user numbers (although this depends upon the operating system in use).

Archive (t3'): designed for user written archive programs. Generally, the archive attribute is set when a file is copied to backup storage, and reset whenever the file is altered. This facility allows a program to determine whether a backup operation is needed. Some operating systems (such as MP/M II) reset the archive attribute of a file automatically upon writing to it.

The attribute value is encoded by reserving one bit position for each of the three attributes, with t1' in the least significant position.

attr	bits	description
0	0 0 0	[no attributes set]
1	0 0 1	read/only
2	0 1 0	system
3	0 1 1	read/only - system
4	1 0 0	archive
5	1 0 1	archive - read/only
6	1 1 0	archive - system
7	1 1 1	archive - system - read/only

Attibutes of a file may be altered with the ATTRS statement, which is defined in section 4.007.

4.007 ATTRS

Format: ATTRS(x$) = x

Type: statement

Purpose: to allow a change of file attributes

Example: ATTRS("NAMES.DAT") = 1

BASIC/Z recognizes three general file attributes: read/only, system, and archive. Specific definitions of each attribute type may be found under the ATTR function, in section 4.006.

The ATTRS statement is used to alter the attributes of a disk file. This is accomplished by passing a numeric argument (x) in the range of zero (0) to seven (7), which corresponds to the desired attribute combination.

The expression x$ must evaluate to the name of the file to be acted upon, which may not be open at the time of execution. The file name must be unambiguous (i.e. contain no ? nor *), and may include an optional drive reference, if located on other than the currently selected default drive.

4.008 BELL

Format: BELL(x)

Type: statement

Purpose: to sound the console bell

Example: IF ERROR.CONDITION& THEN BELL(15)

The BELL statement is used to sound the console bell, generally to gain the operator's immediate attention.

Upon execution, the bell is sounded x times, with a pre-determined delay (configurable at installation) between each. The expression x must evaluate to a numeric value in the range of 0 through 255.

4.009 BIN$

Format: BIN$(x)

Type: intrinsic function

Example: X$ = BIN$(X%)

The BIN$ function returns a sixteen character string which contains the binary representation of the argument x. The expression x must evaluate to a numeric value in the range of 0 to 65535.

4.010 BLINK

Format: BLINK

Type: statement

Purpose: to cause console output to flash

Example: BLINK

Execution of the BLINK statement causes subsequent printed output, directed to the console, to flash on/off, assuming this feature is supported by the console hardware.

see also: NOBLINK (section 4.121)

4.011 CCOL

Format: CCOL

Type: intrinsic function

Example: IF CCOL > 60 THEN PRINT

The CCOL function returns the current column position of the cursor on the system console.

4.012 CHAIN

Format: CHAIN x$

Type: statement

Purpose: to load and execute a BASIC/Z object program, passing variables declared as COMMON

Example: CHAIN "C:PROGRAM2"

The CHAIN statement is a powerful tool which allows construction of programs much larger than system memory size would normally permit. It makes it possible to transfer control and data from section to section of a program which has been divided into separate segments.

BASIC/Z provides three statements which transfer control to another program segment: CHAIN, LINK, and RUN. While similar in some respects, there are fundamental differences, making each more or less suitable for a particular application. In order to execute a CHAIN statement, the following mandatory conditions must be met:

1) The program accepting control must be of type .BZO.
2) Both programs must contain identical COMMON statements.
3) Both programs must contain identical SIZES statements (or defaults).

Other than the above requirements, the only difference CHAIN and RUN is that with CHAIN, the values of all variables declared COMMON are passed to the receiving program. All other arrays are erased, all other scalar numeric variables are reset to zero (0), and all other scalar strings are reset to null strings.

Upon acceptance of control by the receiving program, the following actions are initiated:

1) Any global error trap (ON ERROR GOTO) is reset.
2) Any local error traps (END or ERROR destinations for files) are reset.
3) Subroutine, function, and expression evaluation stacks are cleared.
4) Data pointer for READ statements is restored to the first data item in the program.
5) All definitions (DEF FA) of entry points to assembly language subroutines are cleared.

continued - -

4.012 CHAIN (continued)

All other data and conditions remain unchanged. Among other items, it should be specifically noted that the direction of the print stream and the value of the current string delimiter are unchanged, as well as the memory end (set by MEMEND) and any assembler subroutines loaded above that address.

Unlike most other languages, files may remain open during execution of a CHAIN or RUN statement, offering a substantial speed advantage. While error traps local to each file are reset, they may be re-established with ON END x GOTO or ON ERROR x GOTO, which are described in sections 4.127 and 4.129.

When executing a CHAIN statement, the argument x$ must evaluate to a string containing an optional drive name (if other than the current default) and the name of a BASIC/Z object program to be executed. The file type (.BZO) is assumed, so it need not be included.

see also: COMMON (section 4.023)

4.013 CHR$

Format: CHR$(x)

Type: intrinsic function

Example: A$ = CHR$(X)

The function CHR$ returns a one-byte string consisting of the character which is ascii code x. The numeric expression x must evaluate in the range of 0 through 255.

4.014 CLEAR

Format: CLEAR

Type: option

Example: OPEN 1 "FILE" CLEAR

CLEAR is an option which may be designated as part of an OPEN statement. It is used to create the condition where the file appears to be logically "empty". While it is syntactically allowed in a CREATE statement, it serves no purpose there.

When used with a random or unfmt file, CLEAR takes precedence over the normal initialization of the GET and PUT pointers, resetting them to one (1). A subsequent indexed GET will generate an end of file condition, and an indexed PUT will write to the first logical record.

When used with a sequential file, the end-of-file pointer is set to record one (1), position one (1), emulating a newly created file. The file may only be written to. Any attempt to read will generate an end of file condition.

see also: OPEN (section 4.134), and file handling (section 6)

4.015 CLINE

Format: CLINE

Type: intrinsic function

Example: IF CLINE > 15 THEN GOSUB @PAUSE.ROUTINE

The CLINE function returns the current line position of the cursor on the system console.

4.016 CLOSE

Format: CLOSE [x] {,x}

Type: statement

Purpose: to terminate i/o to a disk file

Example: CLOSE 1, 2, 3, 4

Closing a file consists of updating the directory to reflect all operations which were performed. As a general rule, all files which are opened in a program should be closed explicitly, even if the file was not extended. Failure to do so may cause a loss of data written to the file. In a multi-user environment, it is mandatory that all files opened be closed.

If the CLOSE statement appears with no arguments, then all open files will be closed. If CLOSE is followed by one or more arguments, each must evaluate to a file number (in the range of 0 through 29), associated with a file to be closed. Multiple file numbers must be delimited by commas (,).

Each time a file is opened, a buffer of approximately 190 bytes is reserved, regardless of the file type or the logical record length. When that file is subsequently closed, the buffer space is generally not released, but rather it is reserved for use by another file to be opened under the same file#. It is therefore most memory efficient to "re-use" file numbers whenever practical. An exception to this rule occurs when a close is executed with no file# arguments. In this case, all open files will be closed, and all reserved file buffer space will be released for dynamic allocation.

4.017 CLRAUTO

Format: CLRAUTO

Type: statement

Purpose: to specify stack reset on an error condition

Example: CLRAUTO

The CLRAUTO statement is used to specify that all run-time stacks be automatically cleared whenever an error condition occurs. As this is the default condition, CLRAUTO need not be executed except to reverse the effect of a CLRNONE statement (section 4.019). Upon execution of a program or a program segment (as with CHAIN or RUN), all run-time stacks are implicitly cleared by the system. The run-time stacks are defined as follows:

Subroutine stack: This stack is used to maintain subroutine return addresses, which may be nested up to 128 levels. It may be reset individually through execution of a CLRSUB statement (section 4.020).

Function stack: This stack is used to maintain return addresses and data buffers for user-defined functions. Nesting is limited only by available system memory. This stack may be reset individually through execution of a CLRFN statement (section 4.018).

Expression stack: This stack is used in evaluation of numeric and string expressions. The maximum size is automatically calculated by BASIC/Z, based upon the most complex expression in your program. This size is reported as "System buffer" in the memory allocation summary, printed at the end of every compilation. This stack may be reset individually through execution of a CLRFN statement (section 4.018).

4.018 CLRFN

Format: CLRFN

Type: statement

Purpose: to clear the function and expression evaluation stacks

Example: CLRFN

The CLRFN statement is used to reset the function stack and the expression evaluation stack. Upon execution of a program or a program segment (as with CHAIN or RUN), all run-time stacks are implicitly cleared by the system. In general, there are two instances where CLRFN would be used:

You may encounter a situation where it is necessary to force an exit from a multi-line user-defined function, rather than returning properly through the FNEND statement. Normally, this would leave "garbage" on the function and expression stacks, which could cause a later program crash. If CLRFN is executed, memory occupied by the function stack is released, and the expression evaluation stack is reset to prevent an overflow on a later calculation. It should be noted that this technique is generally considered a poor programming practice, and should probably be avoided.

If CLRNONE has been executed to inhibit stack reset on an error condition, your global error trap routine should usually contain a CLRFN statement to reset the function and expression stacks. Generally, the only exception to this rule would be the case where your error trap is included within the same multi-line user-defined function as the line which generated the error.

4.019 CLRNONE

Format: CLRNONE

Type: statement

Purpose: to inhibit stack reset on an error condition

Example: CLRNONE

The CLRNONE statement is used to specify that run-time stacks should not be automatically cleared when an error condition occurs. The default condition of automatic clearing may be restored at any time by execution of a CLRAUTO statement (section 4.017).

Under certain conditions, it is desirable to locate an error trap within the same subroutine or user-defined function as the line which causes the error, as in the following example:

```
100 gosub @file.dump
110 end
120 '
130 @file.dump
140 clrnone
150 open 1 "FILE.DAT" end @file.dump.end
160 @file.dump.1
170 getseq 1 text$
180 print text$
190 goto @file.dump.1
200 '
210 @file.dump.end
220 close 1
230 return
```

In the above example, the subroutine @file.dump merely reads and prints a sequential file until an end file error is generated. With many languages, this would be impossible, as the error would cause the subroutine stack to be cleared, thus generating another error at line 230. However, with BASIC/Z, you have full control over all aspects of error handling.

Whenever the CLRNONE statement is used, it is most important that you analyze the error handling most carefully. Caution must be exercised to avoid leaving "garbage" on the subroutine or function stacks. When invalid data remains, you run a very real risk of a program crash through a stack overflow, or an out of memory condition. Stacks may be reset with CLRFN (section 4.018) or CLRSUB (section 4.020).

4.020 CLRSUB

Format: CLRSUB

Type: statement

Purpose: to clear the subroutine stack

Example: CLRSUB

The CLRSUB statement is used to reset the subroutine return address stack. Upon execution of a program or a program segment (as with CHAIN or RUN), all run-time stacks are implicitly cleared by the system. In general, there are two instances where CLRSUB would be used:

You may encounter a situation where it is necessary to force an exit from a subroutine, rather than returning properly through the RETURN statement. Normally, this would leave invalid return addresses on the subroutine stack, which could later cause an overflow and subsequent program crash. Execution of CLRSUB will reset the subroutine stack to avoid this problem. It should be noted that this technique is generally considered a poor programming practice, and should probably be avoided.

If CLRNONE has been executed to inhibit stack reset on an error condition, your global error trap routine should usually contain a CLRSUB statement to reset the subroutine stack. Generally, the only exception to this rule would be the case where your error trap is included within the same subroutine as the line which generated the error.

4.021 CLS

Format: CLS

Type: statement

Purpose: to erase the terminal screen

Example: CLS

Execution of the statement CLS will cause the terminal screen to be erased, regardless of the direction of the print stream.

While the FORMFEED statement (section 4.071) will also erase the screen, it requires that the print stream be directed to the console.

4.022 COMMAND$

Format: COMMAND$

Type: intrinsic function

Purpose: to retrieve a command line trailer passed to a program

Example: IF COMMAND$ <> "PASSWORD" THEN STOP

The COMMAND$ function returns a string of up to ten (10) characters, which were typed as a command line trailer at execution of the compiled program. This allows a password or special program directive to be passed by the operator.

Upon execution of your program, the command line will be scanned for any additional text following the name of the program. If any is found, it will be preserved for future recall with the COMMAND$ function. Any characters in excess of ten (10) will be discarded.

The command line trailer may optionally begin with an ampersand (&) as a delimiter. However, a leading ampersand, if present, is never passed to the COMMAND$ function.

4.023 COMMON

Format: COMMON list of variables

Type: statement

Purpose: to define the variables passed to or from a chained program

Example:
```
100 COMMON A, B%, C$, D#(5,8), E$(55)
200 COMMON F, G%, H$, I#(5,8), J$(55)
```

The COMMON statement is used to define those variables which are to be COMMON to two or more program segments accessed by CHAIN statements. This allows construction of programs much larger than system memory size would normally permit, since control and data can easily be passed from segment to segment.

One or more COMMON statements must appear in every program which initiates a CHAIN, or accepts control via a CHAIN statement. They define which variables will be passed from segment to segment. COMMON statements in the initiating program and the receiving program may contain different variable names, but they must be identical as respects the number of variables passed, and the sequence of variable types. In the above example, assume that line 100 appears in the initiating program, and line 200 appears in the receiving program. Upon execution of a CHAIN, the value of A would be assigned to F in the receiving program, the value of B% would be assigned to G%, etc.

Unlike many other languages, BASIC/Z allocates string space statically. That is, the memory location of a string variable does not generally change during execution of a program, regardless of how many times the value is altered. This offers a substantial speed advantage, as "garbage-collection" delays are eliminated. However, it imposes an additional burden on the programmer, as every string variable (both scalar and array) must have a defined maximum length, which may not be exceeded.

BASIC/Z implies a default maximum string length (known as SSIZE) of forty (40) characters for any scalar (non-array) string variable. If a SIZES statement is included in your program, then the third argument designates an explicit SSIZE, in the range of 1 through 250, which takes precedence over the implied default.

continued - -

4,023 COMMON (continued)

A scalar string may be declared COMMON in either of two ways, as shown in the following example:

```
100  COMMON  TEST$,  TEST.LONG$(250)
```

Both of the above declarations reference scalar (non-array) strings. TEST$ is allowed a maximum legth equal to the default SSIZE, while TEST.LONG$ has a maximum length of 250 characters. Regardless of the method of declaration, any reference to these variables would be by the names TEST$ and TEST.LONG$.

A string array is declared COMMON by including an additional index, to specify the maximum length of each element:

```
100  COMMON  TEST$(9,85)
```

The above statement would be used to declare a COMMON array of ten (10) string elements, each with a maximum length of 85 characters. Any reference to them in the program would be by the name TEST$(x), with x in the range of 0 to 9.

```
100  COMMON  TEST.TWO$(9,9,14)
```

The above statement would be used to declare a COMMON array of one hundred (100) string elements, each with a maximum length of 14 characters. Any reference to them in the program would be by the name TEST.TWO$(x,y), with x and y each in the range of 0 to 9. String arrays may have from one (1) to three (3) dimensions (plus, of course, a final length argument), and are always indexed to zero.

```
100  COMMON  TEST.THREE#(9,9,9)
```

The above statement would be used to declare a COMMON array of one thousand (1000) numeric elements. As all types of numeric arrays have a pre-determined size per element, all of the indexes are accepted as array dimensions. Any reference to these array elements would be by the name TEST.THREE#(x,y,z), with x, y, and z each in the range of 0 to 9. Numeric arrays may have from one (1) to four (4) dimensions, and are always indexed to zero.

In the case of COMMON arrays, as well as explicitly dimensioned scalar strings, note that the COMMON statement acts as a substitute for the DIM statement, so they may not be repeated. A string or array which is declared COMMON may not be dimensioned dynamically (i.e. the indexes must appear as literal numeric constants, not as variables or expressions).

see also: CHAIN (section 4.012)

4.024 CONSOLE

Format: CONSOLE

Type: statement

Purpose: to restore printed output to the console only

Example: CONSOLE

The CONSOLE statement is used to designate printed output to the console. After execution of CONSOLE, all subsequent output to the print stream (primarily PRINT statements), is directed to the console only. CONSOLE mode is the default condition, and need not be explicitly initialized.

see also: ECHO (section 4.046), LPRINTER (section 4.110), NULL (section 4.123), and SPOOL (section 4.196)

4.025 COS

Format: COS(x)

Type: intrinsic function

Example: A = COS(X)

The COS function returns a numeric value equal to the cosine of the numeric expression x.

The argument x must be expressed in radians. Degrees may be converted to radians by dividing the number of degrees by 2pi, or with the function RAD, described in section 4.160.

4.026 CREATE

Format: CREATE x x$ [options]

Type: statement

Purpose: to create a new disk file

Example: CREATE 1 "A:DATAFILE" ERROR @ERR.HANDLER UNLOCKED

The statement CREATE is used to create a new disk file, which is left open for subsequent disk i/o. The numeric expression x (in the range of 0 through 29) specifies a file number, which will be associated with this file until it is closed. The expression x$ must evaluate to a string containing an optional drive name (if other than the current default) and the name of the file to be created. Options may be chosen, as needed, from the selection below. File lock options (LOCKED, READONLY, UNLOCKED) may not be combined. If none are chosen, a LOCKED file is assumed. File lock options are ignored in a single user system.

It is mandatory that all newly-created files be explicitly closed. This is particularly vital with random files, as failure to do so will leave it without a header record. Any subsequent attempt to open the file would generate a reclen error, rendering the file useless.

OPTIONS :

CLEAR - The CLEAR option is used to create the condition of a logically empty file, by resetting the PUT and EOF pointers in an OPEN statement. Although this option is syntactically allowed, there is no need to include it, as CREATE implies an empty file by definition.

END - The END option is used to designate a destination to GOTO upon an attempt to read past the logical end of this file. It must be immediately followed by a valid line number or label.

ERROR - The ERROR option is used to designate a destination to GOTO if a disk error occurs in a reference to this file. It must be immediately followed by a valid line number or label.

LOCKED - Writing to this file, by other users, is prohibited. If a file is not specified as READONLY or UNLOCKED, then a default to LOCKED will be assumed.

continued - -

4.026 CREATE (continued)

READONLY - The file may be shared, on a read/only basis, with other users. Note that, by definition, this option is not allowed, as CREATE is a write operation. This is valid as an option to an OPEN statement, and is included here for the sake of completeness only.

RECLEN - If RECLEN is not included, the file is assumed to be sequential type. If present, it must be immediately followed by a numeric expression which evaluates to the desired logical record length for a random or unfmt file. This must be greater than zero, and smaller or equal to 250 or the reclen limit specified in a SIZES statement.

UNFMT - UNFMT specifies a simplified form of random access file (a RECLEN option is mandatory) which eliminates the header record. This allows compatibility with other languages, but may introduce certain errors in the PUT and EOF pointers when the logical record length is set to less than 128. Also, in the UNFMT mode, the logical record length and logical file size (if altered with the EOF statement) are not maintained by the system, but just discarded after the file is closed.

UNLOCKED - The file may be shared, on a read/write basis, with other users.

see also: OPEN (section 4.134) and file handling (section 6)

4.027 CVTC

Format: CVTC(x$)

Type: intrinsic function

Example: X& = CVTC(X$)

The function CVTC is used to convert the first byte of the string expression x$, in the internal control variable format (one byte unsigned binary), into a numeric value. It returns a numeric value in the range of 0 through 255.

see also: control variables (section 2.010)

4.028 CVTC$

Format: CVTC$(x)

Type: intrinsic function

Example: X$ = CVTC$(X&)

The function CVTC$ is used to convert the value of the numeric expression x, into a one byte string, which is equivalent to the internal control variable format (one byte unsigned binary) of the value. The numeric expression x must evaluate in the range of a control variable (0 to 255).

see also: control variables (section 2.010)

4.029 CVTCD

Format: CVTCD(x$)

Type: intrinsic function

Example: X# = CVTCD(X$)

The function CVTCD is used to convert the first two bytes of the string expression x$, in the internal control/d variable format (two byte unsigned binary - low/high), into a numeric value. It returns a numeric value in the range of 0 through 65535.

see also: control/d variables (section 2.011)

4.030 CVTCD$

Format: CVTCD$(x)

Type: intrinsic function

Example: X$ = CVTCD$(X#)

The function CVTCD$ is used to convert the value of the numeric expression x, into a two byte string, which is equivalent to the internal control/d variable format (two byte unsigned binary - low/high) of the value. The numeric expression x must evaluate in the range of a control/d variable (0 to 65535).

see also: control/d variables (section 2.011)

4.031 CVTI

Format: CVTI(x$)

Type: intrinsic function

Example: X% = CVTI(X$)

The function CVTI is used to convert the first ISIZE bytes of the string expression x$, in the internal bcd integer variable format, into a numeric value. It returns a numeric value in the range of from 1-5E(2*ISIZE-1) to 5E(2*ISIZE-1).

see also: integer variables (section 2.012)

4.032 CVTI$

Format: CVTI$(x)

Type: intrinsic function

Example: X$ = CVTI$(X%)

The function CVTI$ is used to convert the value of the numeric expression x, into an ISIZE byte string, which is equivalent to the internal bcd integer format of the value. The numeric expression x must evaluate in the range of a bcd integer variable: 1-5E(2*ISIZE-1) to 5E(2*ISIZE-1).

see also: integer variables (section 2.012)

4.033 CVTR

Format: CVTR(x$)

Type: intrinsic function

Example: X = CVTR(X$)

The function CVTR is used to convert the first RSIZE bytes of the string expression x$, in the internal real variable (i.e. floating point) format, into a numeric value. It returns a numeric value in the range of (-1E+63)+1 to (1E+63)-1.

see also: real variables (section 2.013)

4.034 CVTR$

Format: CVTR$(x)

Type: intrinsic function

Example: X$ = CVTR$(X)

The function CVTR$ is used to convert the value of the numeric expression x, into an RSIZE byte string, which is equivalent to the internal real variable (i.e. floating point) format of the value. The numeric expression x must evaluate in the range of a real variable: (-1E+63)+1 to (1E+63)-1.

see also: real variables (section 2.013)

4.035 DATA

Format: DATA list of constants

Type: statement

Purpose: to store data items which may be accessed by a READ statement

Example: DATA 3,"April 24, 1946",33

DATA statements are not executed, but define a list of constants which may be assigned to variables using a READ statement. Any number of DATA statements can be located anywhere in a program, but BASIC/Z treats them as one list of constants available at run-time.

Upon program execution, the DATA pointer is initialized to point to the first item in the first DATA statement in the program. When a READ statement is executed, one value is read from the list for each variable specified, and the pointer is advanced to the next data item. When the data items in a DATA statement are depleted, the pointer is positioned at the next DATA statement in the program, such that all data items constitute a contiguous list. The DATA pointer may be re-positioned at any time with the RESTORE statement.

The DATA statement must be the only statement on a line. A colon (:) on a DATA line is considered to be part of a literal constant, rather than a statement separator, so that even REM statements are illegal. For example:

```
100 DATA JOHN : REM the first name
200 READ X$
```

Upon execution of the above, X$ would equal 'JOHN : REM the first name', rather than just 'JOHN' as might be expected.

Data items may be either string or numeric. A READ of a numeric data item follows the rules of the VAL function. That is, the following two examples produce the identical end result:

```
READ  X        (or)        READ X$ : X = VAL(X$)
```

Each DATA item in the list of constants must be delimited by a comma (,). If a string DATA item contains a comma (,), or leading spaces are significant, the entire string must be enclosed in quotes (").

Every field must contain a significant data item:

```
100 DATA JOHN,,MARY
110 DATA JOHN,"",MARY
```

Line 100 above is illegal as no data item appears between the two commas. A null string may be represented as a data item by including two adjacent double quotes, as shown in line 110 above.

see also: READ (section 4.162) and RESTORE (section 4.173)

4.036 DEBUG

Format: DEBUG type, start, end {,var}

Type: statement

Purpose: to specify desired debugging options

Example:
```
DEBUG "T", @LIST.IT, @NO.LIST
DEBUG "S", 500, 800, A%, B%, C$, D$
```

DEBUG statements are used to invoke several unique options, to aid you in testing and debugging your programs. DEBUG statements are not executed, but rather declare a requested option and may appear at any place in a program. DEBUG statements are only recognized if your program is compiled with the DEBUG option (&D) invoked in the option parameter. Therefore, there is no need to remove them upon "final" compilation, as they will be ignored.

A program may contain up to sixteen (16) DEBUG statements, and they may be intermixed by type. This allows you to concentrate your efforts on specific sections of code. In every DEBUG statement, the first three parameters (type, start, end) are mandatory. Every parameter must be delimited by a comma (,).

Generally, executing programs may not be paused nor interrupted, to help prevent an unsophisticated user from "crashing" a critical program. However, in the DEBUG mode, it is assumed that a knowledgeable operator is controlling the system. Therefore, when executing that part of a program designated by a DEBUG statement, control-s may be used to pause and restart execution, while control-c may be used to cancel execution and return to the operating system.

The parameter "type" specifies the chosen DEBUG option. It must appear as one or two literal letters, enclosed in double quotes ("). Only one option may be specified in each DEBUG statement.

The parameters start and end specify the range of BASIC/Z statements affected by this option. Each may be specified by your choice of a line number or a label, but the referenced items must be present in the program. Optionally, either, or both, may be replaced with an asterisk (*), to reference the first and/or last statement in the program. For example, DEBUG "T", *, * would specify a trace on the entire program.

Of course, to gain this added function, a price must be paid in terms of generated object code, although only over the affected areas. The "Lines" option generates 6 bytes of added code for each affected program line, while "Pause" adds 3 bytes for each BASIC/Z statement. The remaining options add an overhead of 6 extra bytes of code for each affected statement.

continued - -

4.036 DEBUG (continued)

DEBUG TYPES:

"L" (Lines) - In this mode, line numbers are compiled into the object program to aid in locating an error. If global error trapping is enabled, the line number of the most recent error may be retrieved with the function ERRLINE. If not, the line number of the error will be displayed along with the error message. Generally, this option should not be specified on just part of the program, as misleading results may be returned. If this option is not included in a program, the ERRLINE function returns a value of zero (0).

"P" (Pause) - In this mode, keyboard entry of control-s may be used to pause and restart execution, while control-c may be used to cancel execution and return to the operating system.

"T" (Trace) - In this mode, as each statement is executed, the line number will be displayed on the console, enclosed in brackets ([]). In the case of multi-statement lines, the line number will be repeated for each statement.

"TP" (Trace/printer) - In this mode, as each statement is executed, the line number will be printed on the printer, enclosed in brackets ([]). In the case of multi-statement lines, the line number will be repeated for each statement.

"S" (Single-step) - In this mode, up to four scalar (non-array) variable names may be appended to the DEBUG statement, each delimited by a comma (,). Each time that a statement is executed within the affected range, the following occurs:

1) program execution is paused
2) the line number is displayed, enclosed in brackets ([])
3) current values of the specified variables are displayed
4) BASIC/Z awaits a keyboard entry
5) if control-c is typed, program execution is terminated
6) if any other key is pressed, program execution continues

By setting the start and end argument to the same statement, this option will simulate a typical "breakpoint" function of debuggers.

It should be noted that all debug line displays react to program statements, rather than program lines. Care should be exercised to avoid a misunderstanding of this facility. For example:

```
100  IF X = Y THEN 200
```

The above example actually consists of two statements, a logical comparison of the variables X and Y, followed by conditional execution of an implied GOTO. In tracing the above statement, [100] would display twice if X = Y, or just once if not. Similarly, single-step would pause twice if the condition is true, or just once if not.

4.037 DECR

Format: DECR nvar [,x]

Type: statement

Purpose: to decrement a numeric variable

Example: DECR COUNTER#(X%)

Execution of a DECR statement causes the value of the numeric expression x to be subtracted from the numeric variable. If x is not included, BASIC/Z supplies a value of one (1). For example:

```
DECR x&, y&
LET x& = x& - y&
```

The above statements are functionally identical, as they both provide the same result. However, the DECR statement is preferable, as it is considerably more efficient in both execution speed and generated object code.

4.038 DEF FA

Format: DEF FAname = x

Type: statement

Purpose: to name and define an entry point address in an assembly language program

Example: DEF FATEST$ = ENDMEM - 16R1000

In BASIC/Z, assembly language subroutines are referenced as user-defined functions. This allows compiled BASIC/Z programs to call assembly language programs that have been loaded into memory. The usual purpose is to perform functions that would be difficult or impossible in BASIC.

The name of an assembler function is FA followed by a valid variable name which agrees in type with the data to be returned (i.e. if the assembler subroutine returns a string, the function name must end with a dollar sign '$'). The numeric expression x specifies an entry address in the assembly language subroutine.

Up to 32 functions may be defined within a single program. The definitions are dynamic, so they may be re-defined, but must be executed prior to any reference.

see also: user-defined functions (section 5)

4.039 DEF FN

Format: DEF FNname [(dummy {,dummy})] [= expr]

Type: statement

Purpose: to name and define a user-written function

Example:
```
DEF FNLINE$(X) = REPEAT$("-",X)

DEF FNTEST(X,Y)
IF X > Y THEN X = X*2
FNEND = X
```

The name of a user-defined function is FN followed by a variable name which agrees in type with the data to be returned (i.e. if the function returns a string, the function name must end with a dollar sign '$'). A function name may be defined only once within a program.

Up to four function parameters are optional. If present, they are dummy parameters, which may take any scalar (non-array) variable name. A function parameter, when used, has the additional attribute of being a place-holder. As such, every occurrence of the function parameter name in the expression will cause BASIC/Z to replace the function parameter name with the actual value appearing in the call, before the user-defined function is evaluated.

see also: user-defined Functions (section 5)

4.040 DEG

Format: DEG(x)

Type: intrinsic function

Example: Y = DEG(X)

The function DEG is provided to convert the numeric expression x, expressed in radians, to degrees.

4.041 DDIM

Format: DDIM list of subscripted variables

Type: statement

Purpose: to dynamically define an array

Example: DDIM A%(C%,3)

The DDIM statement is used to dynamically dimension an array. Unlike the standard DIM statement, memory is allocated at run-time, which offers several advantages. The indexes in a DDIM statement are not limited to constants, but may be variables, or even complex expressions. Further, the arrays may be deleted with the ERASE statement (section 4.055), to reclaim the memory space. Of course there are disadvantages also, including the time needed to execute the memory allocation, and the inability to declare them as COMMON data.

When DDIM is used to define a numeric array, the name consists of a valid variable name, optionally followed by an ampersand (&), pound sign (#), or percentum (%), to designate an array of other than real type. A numeric array may have one to four dimensions, which are always indexed to zero (0).

```
100  ddim  test%(a%,b%)
```

The above statement would be used to define an array of (a%+1)*(b%+1) numeric elements. Any reference to the individual array elements would be by the name test%(x,y), with x in the range of 0 to a%, and y in the range of 0 to b%.

When DDIM is used to define a string array, the name consists of a valid variable name, followed by a dollar sign ($). The array may have one to three dimensions, which are always indexed to zero (0).

In BASIC/Z, every string variable must have a defined maximum length, which may not be exceeded. This is specified by including an additional index, to define the maximum length of each element, in the range of 1 through 250.

```
100  ddim  test$(a%,b%)
```

The above statement would be used to define an array of a%+1 string elements, each with a maximum length of b% characters. Any reference to the individual array elements would be by the name test$(x), with x in the range of 0 to a%.

see also: DIM (section 4.043)

4.042 DETACH

Format: DETACH

Type: statement

Purpose: to release the attached printer

Example: DETACH

In a multiple printer system, the DETACH statement is used to release the printer currently attached to this process. The precise implementation of this statement will depend upon the operating system in use. Specific information on the methods used will be found under the ATTACH function, in section 4.005.

When running under an operating system whih does not support multiple printers, execution of the DETACH statement is disabled.

4.043 DIM

Format: DIM list of subscripted variables

Type: statement

Purpose: to define an array, or vector, of data items, to allow easy computations to be performed on tables of data. It may also be used to designate the maximum length of a scalar string variable

Example: DIM A(20), B%(12,16), C$(250)

A DIM statement is used to define an array of data items, and to declare the maximum length of a scalar (non-array) string variable. DIM statements are scanned at compile time, rather than executed. This pre-allocation of memory offers maximum efficiency, as no code is generated. For this reason, however, all indexes in the DIM statement (although not references to array elements) must be expressed as literal constants, rather than variables or expressions. When literal indexes are not practical, the DDIM statement (section 4.041) should be used in place of DIM.

Unlike some other dialects of BASIC, an implied array dimension of ten is not recognized by BASIC/Z. Every array must be explicitly delared with DIM, DDIM, or COMMON.

continued - -

4.043 DIM (continued)

When DIM is used to define a numeric array, the name consists of a valid variable name, optionally followed by an ampersand (&), pound sign (#), or percentum (%), to designate an array of other than real type. A numeric array may have one to four dimensions, which are always indexed to zero (0).

```
100  dim  test%(9,9)
```

The above statement would be used to define an array of one hundred (100) numeric elements. Any reference to the array elements would be by the name test%(x,y), with x in the range of 0 to 9, and y in the range of 0 to 9.

Unlike many other languages, BASIC/Z allocates string space statically. That is, the memory location of a string variable does not generally change during execution of a program, regardless of how many times the value is altered. This offers a substantial speed advantage, as "garbage-collection" delays are eliminated. However, it imposes an additional burden on the programmer, as every string variable (both scalar and array) must have a defined maximum length, which may not be exceeded.

BASIC/Z implies a default maximum string length (known as SSIZE) of forty (40) characters for any scalar (non-array) string variable. If a SIZES statement is included in your program, then the third argument designates an explicit SSIZE, in the range of 1 through 250, which takes precedence over the implied value.

A scalar string which is not declared in a DIM statement has a maximum length equal to SSIZE. If explicitly dimensioned, the maximum length may be set in the range of 1 through 250 characters.

```
100  dim  test$(200)
```

The above scalar string has a maximum length of 200 characters. The subscript is included only to define the length, and is not part of the variable name. All reference to this variable is by the name test$.

When DIM is used to define a string array, the name consists of a valid variable name, followed by a dollar sign ($). A string array may have one to three dimensions, which are always indexed to zero (0). The maximum length of each array element is specified by including an additional index, in the range of 1 through 250.

```
100  dim  test$(29,44)
```

The above statement would be used to define an array of thirty (30) string elements, each with a maximum length of 44 characters. Any reference to the array elements would be by the name test$(x), with x in the range of 0 to 29.

4.044 DISPLAY

Format: DISPLAY x$

Type: statement

Purpose: to display a disk directory

Example: DISPLAY "B:*.BZ?"

The DISPLAY statement is used to output a formatted listing of disk file names to the print stream. Then expression x$ must evaluate to an optional drive and file name mask which is used to limit the file names output. The mask string may be ambiguous - a question mark (?) will match any character, and an asterisk will match any name or type. If a drive name is not included, the currently selected default drive will be assumed.

4.045 DO

Format: DO

Type: statement

Purpose: to repeatedly execute a series of statements, as long as the expression following UNTIL is false

Example:
```
DO
READ ITEM(INDEX&)
INCR INDEX&
UNTIL INDEX& > 20
```

BASIC/Z provides three looping structures, to allow conditional repetition of a series of statements. DO and UNTIL statements are always used together, and provide a means to execute the series of statements one or more times. While this structure is quite similar to a WHILE/WEND loop, note that with DO/UNTIL, the statements within the loop are always executed at least once.

Upon execution of UNTIL, the expression following it is evaluated. If the result is false, control is transferred to the first executable statement following the DO. If the result is true, control is transferred to the first executable statement following the UNTIL.

DO/UNTIL loops may be nested up to 255 levels.

see also: FOR (section 4.070) and WHILE (section 4.226)

4.046 ECHO

Format: ECHO [x]

Type: statement

Purpose: to direct printed output to the console and a second device

Example: ECHO

The ECHO statement is used to designate printed output to the console, and a second device simultaneously. After execution of ECHO, all subsequent output to the print stream (primarily PRINT statements), is directed to these two devices.

If the expression x is not present, the print stream will be echoed to the printer. If it is included, it must evaluate to the file number (in the range of 0 to 29) of a sequential file to which the print stream will be spooled.

see also: CONSOLE (section 4.024), LPRINTER (section 4.110), NULL (section 4.123), and SPOOL (section 4.196)

4.047 EDIT$

Format: EDIT$(x) or EDIT$(x$,x[,y])

Type: intrinsic function

Example: REPLY$ = EDIT$(REPLY$,20)

The function EDIT$ is included in BASIC/Z to provide a sophisticated, yet easy to use, means of accepting and editing console input. As an alternative to the typical INPUT statement, it can add immeasurably to program performance. With EDIT$, you may specify, and even display, a maximum input length. Your user has the ability to move the cursor non-destructively through the line, toggle between insert/change modes, and even to temporarily exit the EDIT$ function (to display a "help" message or ?) and re-enter the line at the same position! Although optional, your program can even supply a beginning value to be edited, with the cursor pre-positioned anywhere within the text.

The optional expression x$ allows definition of a beginning value to be edited. It may contain only printable ascii characters, and be from 0 to 250 characters in length. In the single-argument version of EDIT$, a value of null ("") is assumed. The argument x defines the maximum input length. It must evaluate in the range of 0 to 250, and be greater or equal to the length of x$. The optional argument y defines the initial relative cursor position within the text, if other than the normal position 0. If included, it must evaluate in the range of 0 to the length of x$.

In this section, numerous references will be made to data stored in the user configuration area of memory, which may be accessed with a PEEK or POKE statement. In all cases, the memory location will be named here as a label, such as @ED~CHAR. The actual address of each location may be found in appendix D of this manual.

The EDIT$ function allows you to specify a maximum input length, as the numeric expression x, which may not be exceeded by the user. Any attempt to type past the logical end of line will be rejected, as characters typed will be ignored, and the cursor will not move. At that position, only control or terminator keys will be recognized. A maximum input length of zero (0) is legal, in which case only <RETURN> or an alternate terminator key will be acknowledged. It is the programmer's responsibility to insure that the maximum input length will not cause "wrap-around" past the rightmost column of the console.

continued - -

4.047 EDIT$ (continued)

Upon execution of EDIT$, the beginning value x$, if any, is displayed, left justified within a field of x "fill characters", to identify the maximum input length. The user may edit, replace, or add to this data by typing printable characters and control keys described later. As a character is typed, the fill character in that position is overwritten. Conversely, as a character is deleted from the end of the line, it is replaced in that position by a fill character.

The default fill character is a period (.), but it may be changed to any other printable ascii character at installation, or by using POKE to change the code at @ED^CHAR from within a program. A good alternate to consider is the underline (_) character (16R5F), if it would not be confused with the cursor on your system. Optionally, display of fill characters may be suppressed entirely. This may be specified at installation, or from within a program by setting @ED^FILL to a non-zero value, and @ED^CHAR to a space (16R20).

If the position argument y is not included, the cursor will be initially placed at position zero (0), the start of the line. If it is present, the cursor will be initially placed y characters from the start of the line, assuming that position is within the area of text specified by the beginning value x$. This feature is useful if your program is able to determine what part of a line requires editing, or for re-entering an EDIT$ function after it was interrupted. A special note: if @ED^PROMPT is set (as described later), it is assumed that the cursor is already located y characters from the start of the line, and EDIT$ will not move it further.

In some cases, it is desirable to suppress display of the beginning value x$, as well as any fill characters. Generally, this would be the case when EDIT$ is re-entered after an interrupt, as this information is already displayed on the screen, and re-writing it would be a distraction. This is accomplished by setting @ED^PROMPT to a non-zero value.

During execution of EDIT$, various control keys and/or function keys are recognized to provide screen-oriented editing, line termination, and interrupt capabilities. Although any or all may be re-defined with INSTALL, the default keys are:

Delete character left:	^H	Line terminator:	<RETURN>
Delete character left (alt1):	^Z	Line terminator (alt01):	ESC
Delete character left (alt2):	DEL	Line terminator (alt03):	^B
Delete entire line:	^X	Line terminator (alt05):	^N
Delete character right:	^C	Line terminator (alt07):	^O
Cursor line start:	^W	Line terminator (alt09):	^P
Cursor word left:	^A	Line terminator (alt11):	^Q
Cursor character left:	^S	Line terminator (alt13):	^R
Cursor character right:	^D	Line terminator (alt15):	^T
Cursor word right:	^F	Line terminator (alt17):	^U
Cursor line end:	^E	Line terminator (alt19):	^Y
Insert/change mode toggle:	^V		

continued - -

4.047 EDIT$ (continued)

For each of the previously defined special functions, BASIC/Z will recognize a key which generates from one (1) to six (6) characters, provided that the first, or only, character must be a control code (0-1F hex), or a DEL code (7F hex).

In the insert mode, each character typed is inserted at the cursor position, with remaining characters "bumped" one position to the right. In the change mode, characters are just overwritten. Upon program execution, the change mode is in effect. However, the status of the insert/change mode toggle is preserved from one execution of EDIT$ to the next. Between execution of two EDIT$ functions, the status of this toggle may be interrogated, or it may be forced to a known state. @ED^INSERT must be zero for change mode, or non-zero for insert mode.

The EDIT$ function is terminated, or interrupted, by pressing <RETURN>, or one of the ten alternate terminator keys, which causes the edited version of the text to be returned to the calling expression. If <RETURN> was pressed, the ALTKEY function is set to zero (0), while an alternate terminator key will set it to a corresponding odd number in the range of 1 through 19. Your program can use this information to move from field to field in a requested direction, print help messages, use default answers, or anything else you can imagine! Since BASIC/Z recognizes an odd number as true, a simple statement such as:

```
100  if altkey then . . . .
```

can be used to determine if any one of the ten alternate keys was pressed. In addition, the TERMPOS function is set equal to the relative position of the cursor in the line, at the time of termination. This value can be used as the position argument y if the EDIT$ is to be re-entered. Examples of these features will be found later in this section.

Normally, upon termination, any fill characters remaining on the screen are erased, and the cursor is positioned at the start of the line. These actions can be suppressed by setting @ED^FINISH to a non-zero value. Generally, this would be desirable if there is a possibility of re-entering the function. Of course, your program must then take care of "tidying up" the screen upon a final termination.

An important point about re-entering EDIT$: save the cursor position! Upon an interrupt, retrieve the CCOL and CLINE and save them first! The cursor must be repositioned there before executing EDIT$ again.

continued - -

4.047 EDIT$ (continued)

Following in this section is an example of a multi-line user-defined function which makes extensive use of the ALTKEY capabilities of the EDIT$ function. This defined function, named fnedit.help$, makes the follwing assumptions:

1) The normal insert/change mode toggle has been disabled with INSTALL.
2) For your purposes, alternate terminator key 01 has been designated as the key to toggle insert/change modes.
3) When the insert mode is active, the message "Insert mode on " should be displayed at line 1, column 64.
4) When the change mode is active, the message "Insert mode off" should be displayed at line 1, column 64.
5) When alternate terminator key 03 is pressed, help message A should be displayed at line 24, column 14.
6) When alternate terminator key 05 is pressed, help message B should be displayed at line 24, column 14.
7) Help messages should be erased as soon as another key is pressed.
8) All other alternate terminator keys should be ignored.
9) The function should terminate only when <RETURN> is pressed.

This example does not purport to anticipate every possibility. For example, the status of the insert/change mode toggle should probably be displayed at the first execution. However, we hope it will give you an idea of the potential available.

continued - -

4.047 EDIT$ (example)

```
def fnedit.help$( text$, max.len& )
linel& = cline : ' save the absolute starting position
coll&  = ccol
poke( setup + ed.prompt#) = 0 : ' print the prompt this time
poke( setup + ed.finish#) = 1 : ' but suppress the finish-up
text$  = edit$( text$, max.len& )
poke( setup + ed.prompt#) = 1 : ' suppress further prompts
'
while altkey : ' only if an alternate key was pressed
  line2& = cline : ' save the latest cursor position
  col2&  = ccol
  '
  when altkey = 1 : ' process a mode toggle
  mode& = peek( setup + ed.insert#)
  if mode& = 0 then mode& = 1 else mode& = 0
  poke( setup + ed.insert#) = mode&
  print tab( 1, 64); "Insert mode ";
  if mode& then print "on "; else print "off";
whend
'
when altkey = 3 : ' process help message a
  print tab(24,14); "This is an explanatory message!";
whend
'
when altkey = 5 : ' process help message b
  print tab(24,14); "This is another explanatory message!";
whend
'
when altkey = 3 or altkey = 5 : ' wait till it's been read
  print tab( line2&, col2& ); :  ' let him think he can type
  do
  until instat :                 ' and just wait until he does
  print tab(24,14); : eraeol :   ' then erase the message
whend
'
print tab( line2&, col2& ); : ' place the cursor at the termpos
  text$ = edit$( text$, max.len&, termpos )
wend
'
print tab( linel&, coll& ); : ' place the cursor at the start
print text$; spc$(max.len& - len(text$)); : ' tidy up the screen
print tab( linel&, coll& );
fnend$ = text$
```

4.048 ELSE

Format: IF x THEN stmt {:stmt} [ELSE stmt {:stmt}]

Type: option

Purpose: to control program flow in the event that the expression in an IF statement evaluates false

Example: IF A=B THEN C=1 ELSE C=0

ELSE is an option which may be designated as part of an IF statement. For a complete description, please refer to IF (section 4.085).

4.049 END

Format: END

Type: statement

Purpose: to terminate execution of a program

Example: END

In BASIC/Z, the END statement is optional. Upon execution, the program will be terminated, and control will be returned to the disk operating system.

A program will automatically terminate after executing the last physical statement in the program, whether or not it is an END statement.

4.050 END

Format: END dest

Type: option

Example: OPEN 1 "TESTFILE" END @CLOSE.ROUTINE

The END option is used in a CREATE or OPEN statement to specify a destination to GOTO upon an attempt to read a record past the logical end of this file. It must be immediately followed by a valid line number or label.

An end of file condition is no more than a special class of system error. Therefore, if an END option is not present, it will be treated as any other error.

Regardless of the method of error trapping involved, an end of file condition will generally cause the subroutine, function, and expression stacks to be reset. The only exception to this rule is in the case where the CLRNONE statement (section 4.019) has been invoked.

see also: CREATE (section 4.026), OPEN (section 4.134), ON END x CLEAR (section 4.126), and ON END x GOTO (section 4.127)

4.051 ENDMEM

Format: ENDMEM

Type: intrinsic function

Example: IF ENDMEM < 16R8000 THEN GOTO @OUT.OF.MEMORY

The function ENDMEM returns a numeric value equal to the address of the highest available memory location at the time of program execution. The value returned is not affected by execution of a MEMEND statement. It may be used in a MEMEND argument to reserve (e.g. MEMEND ENDMEM-1000) or to restore (e.g. MEMEND ENDMEM) a memory block.

see also: MEMEND (section 4.113)

4.052 EOF

Format: EOF(x) = y

Type: statement

Purpose: to set the logical end of a file

Example: EOF(1) = 50

The EOF statement is used to alter the logical size of a random, or unfmt file. Upon execution, file #x will be considered to contain y records. Any attempt to read a record greater than y will generate an end of file error.

The file number expression x must evaluate in the range of 0 throuth 29, while the record number expression y must evaluate in the range of 0 to 65535. The EOF statement has no effect upon the GET and PUT pointers. They may be altered with GETSEEK (section 4.077) and PUTSEEK (4.155).

Execution of an EOF statement with a random file is treated as a permanent change, as this information is recorded in the header record. Any subsequent file access will therefore reflect this change of logical file size.

It should be noted that any use of EOF with an unfmt file is just effective during a single session with the file, as the operating system provides no means to permanently record this data.

4.053 ERAEOL

Format: ERAEOL

Type: statement

Purpose: to erase from the cursor position to end of line

Example: ERAEOL

Execution of the ERAEOL statement will cause the console screen to be erased from the present cursor position, to the end of the current line (assuming the console hardware supports this feature).

The ERAEOL statement is only operative when the print stream is directed to the console device. In any other case, it is ignored.

4.054 ERAEOS

Format: ERAEOS

Type: statement

Purpose: to erase from the cursor position to end of screen

Example: ERAEOS

Execution of the ERAEOS statement will cause the console screen to be erased from the present cursor position, to the end of the screen (assuming the console hardware supports this feature).

The ERAEOS statement is only operative when the print stream is directed to the console device. In any other case, it is ignored.

4.055 ERASE

Format: ERASE [avar] {,avar}

Type: statement

Purpose: to erase an array and re-claim the memory space

Example: ERASE A%(), B$()

The ERASE statement may be used to erase any array which has been dynamically dimensioned (with DDIM), thus releasing the occupied memory for other dynamic allocation.

If only the statement ERASE appears, then all dynamically dimensioned arrays will be erased.

If the statement ERASE is followed by one or more array variable names, then only those arrays will be erased. In this version of the statement, the variable names must appear without any indexes, as in the above example.

Upon execution of ERASE, the assigned address of other dynamic arrays may be altered. In this instance, some caution must be used to insure that it does not invalidate a previous VARPTR reference.

see also: DDIM (section 4.034).

4.056 ERR

Format: ERR

Type intrinsic function

Example: IF ERR=1 THEN PRINT ERR$: GOTO 500

ERR returns a numeric value to specify the error code associated with the most recent error. It may be a system error, in the range defined in appendix C, or a user error, generated by SETERROR (section 4.187). The error code is not reset by a successful operation, and is therefore meaningless unless an error actually occurs.

Please refer to appendix C for a detailed definition of each system error code.

4.057 ERR$

Format: ERR$

Type: intrinsic function

Example: IF ERR=1 THEN PRINT ERR$: GOTO 500

The ERR$ function returns an error message string associated with the most recent error. If it is a system error, in the range defined in appendix C, it will contain a description of the error such as "Disk i/o error". If it is a user error, generated by SETERROR (section 4.187), the string "Error" will be returned. The error code is not reset by a successful operation, and is therefore meaningless unless an error actually occurs.

Please refer to appendix C for a detailed definition of each system error code.

4.058 ERRADDR

Format: ERRADDR

Type: intrinsic function

Example: PRINT ERR$; " at "; ERRADDR : END

The ERRADDR function returns a numeric value equal to the program address at which the most recent run-time error occurred. By comparing this value to a compiler listing which includes the object code option, the statement which caused the error may be found. This method of locating a program problem may be preferable as it does not require the overhead of the additional code generated by DEBUG statements.

4.059 ERRLINE

Format: ERRLINE

Type: intrinsic function

Example: PRINT ERR$; " at line"; ERRLINE

The ERRLINE function returns a numeric value equal to the line number in which the most recent run-time error occurred.

This function is operative only when a program is compiled with the "L" DEBUG option invoked, and only if the error occurred within the area of the program specified by the option. In any other case, an incorrect value will be returned.

4.060 ERROR

Format: ERROR dest

Type: option

Example: OPEN 1 "TESTFILE" ERROR @ERR.HANDLER

The ERROR option is used with a CREATE or OPEN statement to designate a destination to GOTO in case of a disk error in referencing this file. It must be immediately followed by a valid line number or label. If the ERROR option is included, it takes precedence over the standard global error trap (ON ERROR GOTO).

Regardless of the method of error trapping involved, an error condition will generally cause the subroutine, function, and expression stacks to be reset. The only exception to this rule is in the case where the CLRNONE statement (section 4.019) has been invoked.

see also: CREATE (section 4.026), OPEN (section 4.134), ON ERROR x CLEAR (section 4.128), and ON ERROR x GOTO (section 4.129)

4.061 EXP

Format: EXP(x)

Type: intrinsic function

Example: A = EXP(X)

The EXP function returns the exponential of x, i.e., the value of the base of natural logarithms (e = 2.71828 ...) raised to the power x.

4.062 FA

Format: FAname [(expr {,expr})]

Type: assembler function call

Example: DATE$ = FADATE$(MONTH&,DAY&,YEAR&)

The name of an assembler function is FA followed by a valid variable name which agrees in type with the data to be returned (i.e. if the function call returns a string, the name must end with a dollar sign '$').

From zero to four arguments may be passed to the assembler subroutine. If included, they must be enclosed in parenthesis '()', each delimited by a comma.

see also: DEF FA (section 4.038), user-defined functions (section 5)

4.063 FILL

Format: FILL(cavar,cavar) = x$

Type: statement

Purpose: to convert successive bytes of a string to control numeric type, and insert them into a control array

Example: FILL(X&(4), X&(22)) = ANYDATA$

FILL is used to assign characters from a string expression to elements of a control array. This is particularly valuable when the array is used as a file buffer, or to simulate a string longer than 250 bytes (see section 2.016). Upon execution, successive elements of the control array (starting with the first cavar and ending with the second cavar) will be set to the ascii value of the corresponding byte of the string expression x$.

If the string expression x$ is exhausted, then the remaining specified elements of the control array are set to null (binary 0). If the length of x$ is greater than the number of elements to be set, it will be considered truncated and ignored.

For example: fill(x&(2), x&(5)) = "AB"

After execution:

x&(2)	= 65	x&(4)	= 0
x&(3)	= 66	x&(5)	= 0

see also: Array variables (section 2.016), STRING$ (section 4.204)

4.064 FILLSPC

Format: FILLSPC(cavar,cavar) = x$

Type: statement

Purpose: to convert successive bytes of a string to control numeric type, and insert them into a control array

Example: FILLSPC(X&(4), X&(22)) = ANYDATA$

The statement FILLSPC is functionally identical to FILL with the exception that if the expression x$ is exhausted, then the remaining affected elements of the control array are set to the ascii equivalent of a space: [chr$(32)].

4.065 FIX

Format: FIX(x)

Type: intrinsic function

Example: A = FIX(X)

The function FIX returns a numeric value equal to the truncated integer part of the numeric expression x.

The difference between the FIX and INT functions is that FIX returns the next higher integer for a negative value (i.e. FIX(-4.5) = -4 while INT(-4.5) = -5).

4.066 FMT

Format: FMT(x,x$)

Type: intrinsic function

Example: X$ = FMT(X,"$$Z9V.99")

The FMT function returns a string consisting of the numeric value x, formatted to the specifications presented by the "format picture" x$. Each character in the string, other than the pointer V, represents one character in the result string.

The following characters are used to specify digit formatting:

9 ~ A digit position, where leading zeroes are output as '0'.

Z ~ A digit position, where leading zeroes are output as spaces.

* ~ A digit position, leading zeroes are output as asterisks.

V ~ Specifies alignment of the decimal point. If a V is not present, it is assumed the decimal point is to be at the right end, which results in truncation of any fractional part of the number.

$ ~ A digit position of the number. If more than one "$" appears, the digit position closest to the first non-zero digit of the number is output as a "$". Leading zeroes are output as spaces.

, ~ A comma appearing before the leading digit is output as a space, dollar sign, or an asterisk, depending upon the context of the string.

All other characters are output unchanged. If the number is too large to fit in the format specified, the entire result string is filled with question marks (?).

Although FMT provides a function similar to PRINT USING in other languages, it is particularly unique in that is is allowable in a context other than just printing. For example:

```
1    sizes(6,3,40,250)
2    telephone  = 3124818085
3    tel.mask$  = "(999)  999-9999"
4    telephone$ = fmt(telephone,tel.mask$)
5    any.date   = 30184
6    date.mask$ = "99/99/99"
7    any.date$  = fmt(any.date,date.mask$)
```

Upon execution of the above:

```
telephone$ = "(312)  481-8085"
any.date$  = "03/01/82"
```

4.067 FN

Format: FNname [(expr [,expr])]

Type: user function call

Example: DATE$ = FNDATE$(MONTH&,DAY&,YEAR&)

The name of a user function is FN followed by a valid variable name which agrees in type with the data to be returned (i.e. if the function returns a string, the name must end with a dollar sign '$').

From zero to four arguments may be passed to the user function. If included, they must be enclosed in parenthesis '()', each delimited by a comma.

see also: DEF FN (section 4.039) and user defined functions (section 5)

4.068 FNEND

Format: FNEND = x

Type: statement

Purpose: to terminate a user-written multi-line numeric function

Example: FNEND = X ^ Y - 2

The statement FNEND is used to terminate a multi-line, user-written, numeric function. The value of the numeric expression x is returned to the calling expression.

see also: user defined functions (section 5)

4.069 FNEND$

Format: FNEND$ = x$

Type: statement

Purpose: to terminate a user-written multi-line string function

Example: FNEND$ = X$+Y$

The statement FNEND$ is used to terminate a multi-line, user-written, string function. The value of the string expression x$ is returned to the calling expression.

see also: user defined functions (section 5)

4.070 FOR

Format: FOR nvar = x TO y [STEP z]

Type: statement

Purpose: to provide a looping structure for repetition of statements

Example: FOR Z = 0 TO 100 STEP 2

The series of instructions to be executed begins with the statement immediately following the FOR statement. It ends with the NEXT statement which contains the same variable as the FOR statement.

The numeric variable is used as a counter. The first expression (shown above as x) is the initial value of the counter. The second expression (shown above as y) is the limit value of the counter. If the optional STEP expression is not present, BASIC/Z will supply a default STEP value of plus one (+1). The values of x, y, and z must all be within the range of the counter variable. For example:

```
100 for index% = start% to end% step -1
```

As control variables have a range of 0 through 255, the above statement is illegal, as the step is out of range.

continued - -

4.070 FOR (continued)

The values of the expressions x, y, and z are computed only once, at the time the FOR statement is executed. The BASIC/Z statements between the FOR statement and the associated NEXT statement will always be executed at least one time.

When the associated NEXT statement is encountered, the STEP value is algebraically added to the value of the counter variable. This value is then compared to the limit value. The counter variable is not updated if the limit is exceeded.

If the STEP value is positive and the above-computed value is less than or equal to the limit value, control is transferred to the next statement following the FOR statement. Otherwise, control passes to the first executable statement following the NEXT statement.

If the STEP value is negative and the above-computed value is greater than or equal to the limit value, control is transferred to the next statement following the FOR statement. Otherwise, control passes to the first executable statement following the NEXT statement.

A set of FOR/NEXT statements may be nested within one or more sets of FOR/NEXT statements. However, the inner loop must be completely enclosed within the outer loop.

A unique feature of BASIC/Z is the static design of the FOR/NEXT loop structure:

1) this allows optimum execution speed
2) loops may be exited at any place/need not "fall through"
3) loops may be re-entered after a temporary exit
4) one FOR may have multiple NEXT's
5) multiple FOR's may have one NEXT
6) counter variable must be a scalar (non-array) variable
7) counter variables are not local to a user-written function. If needed, a WHILE/WEND or DO/UNTIL loop should be used instead.

see also: DO (section 4.045) and WHILE (section 4.226)

4.071 FORMFEED

Format: FORMFEED

Type: statement

Purpose: to send a "clear-page" to the print stream

Example: IF LINE.POS& > 56 THEN FORMFEED

The FORMFEED statement is used to send a "clear-page" command to the print stream. If the print stream is directed to the console, the CRT screen will be erased. If the print stream is directed to the printer, it will be set to "top-of-form".

see also: CLS (section 4.021)

4.072 FRAC

Format: FRAC(x)

Type: intrinsic function

Example: A = FRAC(X)

FRAC returns the fractional part of the numeric expression x, with the original sign preserved.

4.073 FREE

Format: FREE(x) or FREE(x$)

Type: intrinsic function

Example: A = FREE(19) or A = FREE("C:")

The FREE function returns a numeric value equal to the number of kilobytes (1024 bytes) of unallocated space on a disk drive associated with file #x, or the disk drive named by x$.

4.074 GET

Format: GET x [RECORD y] var {,var}

Type: statement

Purpose: to read data from a random or unfmt disk file

Example:
```
GET 1 A,B,C$
GET 1 RECORD 10 A,B,C$
```

The GET statement is used to read one logical record from random or unfmt file number x, and to assign the data read to one or more variables. If the RECORD option is included, the record accessed will be that specified by the numeric expression y. If the RECORD option is not included, the record accessed will be that specified by the indexed GET pointer, which will then be incremented, after the read. If the record number to be accessed is greater than the logical file size, an end of file error will be generated.

All four random disk read statements (GET, GETNUM, GETVEC, and GETVEC$) access the file and read a logical record in the same manner. The difference between them is the nature of the data they expect, and the manner in which this data is parsed into variables.

Generally speaking, the GET statement expects the data to be in ascii format. If more than one variable is specified in the variable list, the record must contain a valid delimiter between each data field. For numeric data, this may be a blank space, while for both string and numeric data the current string delimiter (defined by the STRING statement) is acceptable. Data items are extracted from the record in much the same way as an INPUT statement, due to the similarity of the expected data. The record is scanned from left to right, until a valid delimiter is reached for each variable. If too few values exist in the record to satisfy the variable list, an insufficient data error is generated.

The concept of a string delimiter for disk files is a powerful one, as it allows you to define variable length sub-fields within a logical record. In many instances, this can reduce some of the wasted space associated with random files. The default string delimiter is the comma (,). However, this may be changed to any other character of the 256 possible ones, or it may be disabled entirely with the STRING statement (section 4.203). A string to be read with GET may contain any of the 256 ascii codes, except the code defined as the current string delimiter. Of course, no such restriction applies when the string delimiter is disabled.

continued - -

4.074 GET (continued)

Generally, the GET statement is used to read disk records which were written with a PUT statement. PUT writes numeric values to disk in ascii format, following the rules outlined in section 2.24, Numeric output formats (leading minus sign or blank, ascii digits, and a trailing blank space). The trailing blank space acts as an automatic delimiter. If the string delimiter is enabled, any delimiter appearing between numeric values will be ignored when read by GET.

The PUT statement never inserts string delimiters between string variables or expressions. For example:

```
100 put 1 record 1 a$; b$; c$
110 get 1 record 1 x$, y$, z$
```

If the above two statements were executed, the entire record (i.e. a$+b$+c$) would be assigned to x$ (unless, of course, the variables contained a string delimiter character - normally to be avoided). As no data would remain for y$ and z$, an insufficient data error would be generated. If we assume that the current string delimiter is a comma (,), the correct code would be:

```
100 delim$ = ","
110 put 1 record 1 a$; delim$; b$; delim$; c$; delim$
120 get 1 record 1 x$, y$, z$
```

Upon execution of the above, x$, y$ and z$ would be set precisely equal to a$, b$ and c$ respectively.

The string delimiter must be chosen with some care. It should be a code which will never, under any circumstances, appear as part of the data itself. A wise choice might be chr$(0), or chr$(16RFF). A somewhat tricky situation arises when you attempt to use GET to read data which was written in the internal numeric format of BASIC/Z, since any of the 256 codes may appear in the conversion process. For example:

```
100 delim$ = chr$(0) : string delim$
110 x = 0 : y = 1
120 rec$ = cvtr$(x) + delim$ + cvtr$(y) + delim$
130 put 1 record 1 rec$
140 get 1 record 1 numeric1$, numeric2$
150 a = cvtr(numeric1$) : b = cvtr(numeric2$)
```

At first glance, the above routine would appear to work properly, writing and then reading back two numerics in the internal real (i.e. floating point) format. Actually, an argument error would be generated in line 150, since numeric1$ is zero (0) bytes in length, when cvtr requires it be at least RSIZE bytes long. The reason for this apparent discrepancy is the fact that the cvtr$ function in line 120 sets the first byte of rec$ to a null (binary 0), which is the correct exponent for the value zero, in real format. Obviously, this code conflicts with the string delimiter, causing untold confusion.

continued - -

4.074 GET (continued)

A better alternative for the problem described previously would be to disable the string delimiter, as shown below:

```
100 sizes(6,4,250,250)
110 string ""
120 x = 0 : y = 1
130 put 1 record 1 cvtr$(x); cvtr$(y)
140 get 1 record 1 rec$
150 a = cvtr(left$(rec$,6)) : b = cvtr(right$(rec$,6))
```

One last point to remember - when the string delimiter is disabled (with a STRING "" statement), a GET statement may have only one string variable in the variable list. Although a record may contain only one undelimited string, it may then contain any of the 256 possible codes.

see also: file handling (section 6)

4.075 GETFIELD

Format: GETFIELD x svar {,svar}

Type: statement

Purpose: to read one or more fields from a sequential disk file

Example: GETFIELD 1 A$, B$

The GETFIELD statement is used to read one or more fields from sequential file number x, assigning the data to one or more string variables. Numeric data must be read as a string, and then converted with the VAL function.

A field is delimited by the first occurrence of the current string delimiter (if any), or the record delimiter. The default string delimiter is the comma (,), but it may be changed to any other code, or disabled entirely, with the STRING statement (section 4.203). When the string delimiter is disabled (with a STRING "" statement), GETFIELD functions exactly as the GETSEQ statement. The record delimiter is assumed to be a cr/lf combination, although the lf character is optional. An end file condition is immediately generated when the end of file character (control-z/1A hex) is read.

The high order bit is not considered significant in recognizing record delimiters. That is, both 0D hex and 8D hex are considered to be the cr record delimiter, while both 0A hex and 8A hex are recognized as the lf character, and are ignored. In all other codes, the high order bit is maintained as a significant part of the data.

GETFIELD may only be used with sequential files. An attempt to access a random or unfmt file will generate a a wrong file type error.

see also: file handling (section 6), GETSEQ (section 4.078), and INKEY$ (section 4.091)

4.076 GETNUM

Format: GETNUM x [RECORD y] nvar {,nvar}

Type: statement

Purpose: to read numeric data from a random or unfmt disk file

Example: GETNUM 1 A,B,C%

The GETNUM statement is used to read one logical record from random or unfmt file number x, assigning the numeric data, in internal format, to one or more numeric variables. If the RECORD option is included, the record accessed will be that specified by the numeric expression y. If the RECORD option is not included, the record accessed will be that specified by the indexed GET pointer, which will then be incremented, after the read. If the record number to be accessed is greater than the logical file size, an end of file error will be generated.

All four random disk read statements (GET, GETNUM, GETVEC, and GETVEC$) access the file and read a logical record in the same manner. The difference between them is the nature of the data they expect, and the manner in which this data is parsed into variables.

The GETNUM statement expects the record to contain numeric data, in the internal binary or bcd format described in section 2.010 through 2.013. This statement is unaffected by the status of the current string delimiter. No delimiters of any type are expected between data items in a record. Rather, multiple data items are packed immediately adjacent to the next, and parsed based upon the known size of each. If too few values exist in the record to satisfy the variable list, an insufficient data error will be generated.

```
100 sizes(6,4,250,250)
110 getnum 1 record 1 x, y, z#
```

```
100 sizes(6,4,250,250)
110 string ""
120 get 1 record 1 rec$
130 x  = cvtr(left$(rec$,6))
140 y  = cvtr(mid$(rec$,7,6))
150 z# = cvtcd(right$(rec$,2))
```

The above two examples provide the identical function, although the first is substantially more efficient. They expect record 1 to contain 14 bytes of significant numeric data, in the specified internal formats.

continued - -

4.076 GETNUM (continued)

Generally speaking, a record to be read with a GETNUM statement is one which was originally written with a PUTNUM statement (section 4.154). Although numeric data types may be intermixed in a record, extreme care must be used to insure that a value is written and read from the same variable type, so that the same format, and number of bytes, will be expected. Also, in the case of integer and real variables, it is mandatory that ISIZE and RSIZE be identical at the time of reading and writing.

GETNUM may only be used with random or unfmt files. An attempt to access a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.077 GETSEEK

Format: GETSEEK(x) = y

Type: statement

Purpose: to explicitly set the indexed GET pointer

Example: GETSEEK(1) = 50

In accessing a random or unfmt disk file with GET, GETNUM, GETVEC, or GETVEC$, the record number may be stated explicitly with the RECORD option, or it may be implied as the current value of the indexed GET pointer. A separate GET pointer is automatically maintained for each file. Each time that a file is opened, or created, the GET pointer is initialized to point to record one (1) of the file. Every subsequent read without a RECORD option utilizes the current value, and then increments the pointer by one (1). This allows sequential access to a random or unfmt file with minimum difficulties.

The GETSEEK statement may be used to reset the GET pointer for file number x to the record specified by the numeric expression y. It must evaluate to a number which is greater than zero, and less or equal to the size of the file.

The current value of the indexed GET pointer may be accessed with the RECGET function (section 4.164).

GETSEEK may only be used with random or unfmt files. An attempt to reference a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.078 GETSEQ

Format: GETSEQ x svar {,svar}

Type: statement

Purpose: to read one or more sequential records

Example: GETSEQ 1 A$, B$

The GETSEQ statement is used to read one or more entire logical records from sequential file number x, assigning the data to one or more string variables. Numeric data must be read as a string, and then converted with a VAL function.

A logical record is not affected by the current string delimiter. However, when it is disabled (with a STRING "" statement), GETFIELD functions exactly as the GETSEQ statement. The record delimiter is assumed to be a cr/lf combination, although the lf character is optional. An end file condition is immediately generated when the end of file character (control-z/1A hex) is read.

The high order bit is not considered significant in recognizing record delimiters. That is, both 0D hex and 8D hex are considered to be the cr record delimiter, while both 0A hex and 8A hex are recognized as the lf character, and are ignored. In all other codes, the high order bit is maintained as a significant part of the data.

GETSEQ may only be used with sequential files. An attempt to access a random or unfmt file will generate a a wrong file type error.

see also: file handling (section 6), GETFIELD (section 4.075), and INKEY$ (section 4.091)

4.079 GETVEC

Format: `GETVEC x [RECORD y] navar, navar`
`GETVEC x [RECORD y] savar, savar, x$`

Type: statement

Purpose: to read data from a random or unfmt disk file, parsing it into successive array elements

Example:
```
GETVEC 1 A#(0), A#(99)
GETVEC 1 A$(4), A$(23), MASK$
```

The GETVEC statement was designed to allow a single statement to read a large number of data items from one random or unfmt disk record in file number x, assigning them to an appropriate number of corresponding array elements. If the RECORD option is included, the record accessed will be that specified by the numeric expression y. If the RECORD option is not included, the record accessed will be that specified by the indexed GET pointer, which will then be incremented, after the read. If the record number to be accessed is greater than the logical file size, an end of file error will be generated.

All four random disk read statements (GET, GETNUM, GETVEC, and GETVEC$) access the file and read a logical record in the same manner. The difference between them is the nature of the data they expect, and the manner in which this data is parsed into variables.

The two array variable arguments specify the first and the last element of the array to be assigned data, while all intervening elements are presumed. They must reference elements of the same array, with the first preceding the second in memory sequence. GETVEC is most frequently used with arrays which are one dimensional. However, with care, it may be used with multi-dimensional arrays as well. As described under Array Variables (section 2.16), arrays are stored in column major order, which must be carefully considered. For example:

```
1) getvec 1 x(0), x(9)
2) getvec 1 a(0,0), a(3,2)
3) getvec 1 a(3,0), a(0,1)
4) getvec 1 a(0,2), a(2,0)
```

Example 1 references 10 consecutive elements in a one dimensional array. The remaining examples all reference a two dimensional array which was dimensioned as a(3,2). Example 2 references all 12 elements of the array, in the sequence they are stored in memory. Example 3 references just 2 elements of the array, as they are located adjacent in memory. Example 4 is illegal, as the second precedes the first in memory sequence.

continued - -

4.079 GETVEC (continued)

There are two general forms of the GETVEC statement, which differ depending upon the type of the data, numeric or string.

NUMERIC ARRAY:

The GETVEC statement expects the record to contain numeric data, in the internal binary or bcd format described in section 2.10 through 2.13. This statement is unaffected by the status of the current string delimiter. No delimiters of any type are expected between data items in a record. Rather, multiple data items are packed immediately adjacent to the next, and parsed based upon the known size of each. If the logical record is exhausted before all of the specified elements are assigned data, remaining array elements will be reset to zero (0). Generally, this form of the statement is used to read data which was originally written with the numeric form of PUTVEC (section 4.157), or possibly with PUTNUM (section 4.154). However, great care must be exercised to insure that data is written and read from the same variable type, so that the same format, and number of bytes, will be expected. Also, in the ease of integer and real variables, it is mandatory that ISIZE and RSIZE be identical at the time of reading and writing.

STRING ARRAY:

This form of the GETVEC statement expects the record to contain string data, in fixed length sub-fields of pre-determined length. This statement is unaffected by the status of the current string delimiter. No delimiters of any type are expected between data items in a record. Rather, they are parsed into successive array elements based upon the ascii code of the byte of the mask x$ which corresponds to the relative position of the array element. For example:

```
mask$ = chr$(2) + chr$(17) + chr$(8) + chr$(8)
getvec 1 a$(3), a$(6), mask$
```

The above example assumes that the record contains a minimum of 35 bytes, the sum of the 4 mask characters. Upon execution, the first 2 bytes of the record would be assigned to a$(3), the next 17 to a$(4), the next 8 to a$(5), and the final 8 to a$(6). If the logical record is exhausted before all of the specified elements are assigned data, remaining array elements will be reset to null. Generally, this form of the statement is used to read data which was originally written with the string form of PUTVEC (section 4.157), or PUTVECSPC (section 4.158).

GETVEC may only be used with random or unfmt files. An attempt to access a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.080 GETVEC$

Format: GETVEC$ x [RECORD y] savar, savar, x$

Type: statement

Purpose: to read string data from a random or unfmt disk file, parsing it into successive array elements

Example: GETVEC$ 1 A$(4), A$(23), MASK$

The GETVEC$ statement is very similar in function to the string form of the GETVEC statement (section 4.079). The only difference is that, as values are assigned to array elements, any trailing nulls (binary 0) in each sub-field are deleted.

This feature can be very useful when reading/writing text in conjunction with the PUTVEC statement (section 4.157), as any nulls it adds as "padding" can be automatically removed.

GETVEC$ may only be used with random or unfmt files. An attempt to access a sequential file will generate a wrong file type error.

4.081 GOSUB

Format: GOSUB dest

Type: statement

Purpose: to execute a series of statements as a subroutine

Example: GOSUB @PRINT.ROUTINE

When a GOSUB statement is encountered, control is transferred to the statement at the destination (label or line number) specified. Program execution continues until a RETURN instruction is reached. At that point, control is returned to the statement immediately following the GOSUB. BASIC/Z will allow nested subroutines. That is, a subroutine may call another subroutine, up to a maximum depth of 128 levels. If that limit is exceeded, a stack overflow error will be generated.

BASIC/Z maintains return addresses on a last-in/first-out (LIFO) stack, known as the subroutine stack. Other statements which can modify this stack include CLRSUB (section 4.020), POP (section 4.147), PUSH (section 4.150), and RETURN (section 4.175).

4.082 GOTO

Format: GOTO dest

Type: statement

Purpose: to transfer control to a statement out of sequence

Example: GOTO 200

When a GOTO statement is encountered, control is transferred to the statement at the destination (label or line number) specified.

4.083 HEX$

Format: HEX$(x)

Type: intrinsic function

Example: A$ = HEX$(A%)

The HEX$ function returns a four byte string which contains the hexadecimal representation of the argument x. The numeric expression x must evaluate in the range of 0 to 65535.

4.084 HIGHINT

Format: HIGHINT

Type: statement

Purpose: to cause high intensity console output

Example: HIGHINT

Execution of the HIGHINT statement causes subsequent printed output, directed to the console, to be high (bold) intensity, assuming that this feature is supported by the console hardware.

see also: LOWINT (section 4.109), and STDINT (section 4.199).

4.085 IF..THEN..ELSE

Format:	IF x THEN stmt {:stmt} [ELSE stmt {:stmt}] IF x THEN dest [ELSE dest]
Type:	statement
Purpose:	to control program flow based upon the result of an expression
Example:	IF A=B THEN C=1 ELSE C=0 IF A=B THEN 220 ELSE 320

An IF statement may be used to provide conditional execution of one or more statements, based upon the result of a logical expression. Any statement subject to this conditional execution must appear within the same physical program line as the IF statement. If the expression evaluates true, the THEN clause is executed. If the expression evaluates false, the THEN clause is ignored, and the ELSE clause, if present, is executed. Each of these clauses may contain multiple statements, on the same line, if they are separated by a colon (:), as a statement terminator.

```
100 if a = b then gosub @accept : gosub @print.it
200 print x$
```

In the above example, if a and b are equal, then both subroutines (@accept and @print.it) will be executed. If a does not equal b, control is transferred to the next physical line (200), and neither subroutine is executed.

In execution of an IF statement, BASIC/Z will recognize an implied GOTO. That is, if either THEN or ELSE are immediately followed by a line number or label, control is transferred to the statement at the specified destination.

If it is necessary to conditionally execute statements which span multiple physical lines, a WHEN/WHEND construct (section 4.224) should be used instead.

4.086 INCHAR$

Format: INCHAR$

Type: intrinsic function

Example: IF INCHAR$ = CHR$(3) THEN @CANCEL.ROUTINE

The INCHAR$ function is used to input individual keystrokes from the console, without echoing them to the screen. If a key has been pressed, it returns a one byte string containing the input character. If no input is ready, it returns immediately with a null string. In contrast, the INKEY$(1) function will wait indefinitely until a key is pressed.

INCHAR$ will recognize any code, whether printable or not. For example, pressing the <RETURN> key will return a string equivalent to chr$(13). Also, as input characters are not echoed, this function may be particularly suited to password entry.

see also: INKEY$ (section 4.090) and INSTAT (section 4.094)

4.087 INCLUDE

Format: INCLUDE slit

Type: directive

Purpose: to allow an additional source program to be included in the compilation

Example: INCLUDE "PROGRAM4"

The INCLUDE directive allows source code in a disk file to be included into the main source program during compilation. That is, all BASIC/Z statements in the specified file will be compiled, in line, at the position of the INCLUDE directive. When the end of file is reached, compilation of any remaining statements in the main source program will continue. This concept allows your programs to be broken up into smaller modules, which may be utilized whenever they are needed. A particularly useful application of INCLUDE is for the declaration of COMMON statements, as one change could affect many programs which utilize CHAIN.

An INCLUDE directive must be the only statement on a physical program line. Also, the argument to INCLUDE must be a string literal, enclosed in double quotes ("), which specifies the name of the source file to be included. As an extension of ".BZS" is required and assumed, it need not be included. An optional drive name may appear, to designate the location of the file. If not present, the default include drive, specified as a compile-time option of &I(dr:), or the currently selected default drive will be utilized.

INCLUDE files may not be nested. That is, a file which has been included may not include yet another file. Also, to avoid the obvious problem of line number conflict, all line numbers within included files are ignored. Any reference to a destination statement in an included file (by a GOTO, GOSUB, DEBUG, RESTORE, PUSH, etc.) must be by label only.

4.088 INCR

Format: INCR nvar [,x]

Type: statement

Purpose: to increment a numeric variable

Example: INCR COUNTER#(X%)

Execution of an INCR statement causes the value of the numeric expression x to be added to the numeric variable. If x is not included, BASIC/Z supplies a value of one (1). For example:

```
incr x&, y&
let x& = x& + y&
```

The above statements are functionally identical, as they both effect the same result. However, the INCR statement is preferable, as it is considerably more efficient in both execution speed and generated object code.

4.089 INDEX

Format: INDEX(x$,y$)

Type: intrinsic function

Example: A = INDEX(X$,Y$)

The INDEX function is used to determine whether a string is a sub-string of a second string, and, if so, its relative position. Upon execution, it searches for the first occurrence of the sub-string y$ within x$, returning a numeric value to indicate the relative position at which a match is found. For example:

```
index( "abcdef", "bcd" )   --->   2
index( "abcdef", "ef"  )   --->   5
```

If x$ is null, or y$ is not a sub-string of x$, then 0 is returned. If y$ is null, 1 is returned.

4.090 INKEY$

Format: INKEY$(x[,y])

Type: intrinsic function

Example:
```
TEXT$ = INKEY$(1)
BYTE$ = INKEY$(1,FILE.NUM&)
```

Generally speaking, INKEY$ is a byte oriented data input function. However, it has two major uses which are functionally and syntactically distinct.

If INKEY$ is referenced with a single argument, it is used to input one or more individual keystrokes from the console, without echoing them to the screen. Upon execution, INKEY$ will pause and wait for x characters to be typed at the console, returning them as a string of x bytes. In contrast, the INCHAR$ function returns immediately, whether input is ready or not.

INKEY$ will recognize any code, whether printable or not. For example, pressing the <RETURN> key will return a byte equivalent to chr$(13). Also, as input characters are not echoed, this function may be particularly suited to password entry.

If INKEY$ is referenced with two arguments, it is used to read the next x bytes from sequential file number y. In this form, it returns raw data from the file, making absolutely no logical interpretations. This function recognizes no string delimiters and treats the record delimiters (cr/lf) and the end of file character (control-z/1A hex) as any other codes. An end of file error is generated only when you attempt to read past the last physical record in the file. This feature allows you to create any desired sequential file structure, which may utilize any of the 256 possible codes.

By using a multi-line user-defined function, it is possible to easily add some logic to this function. In the following example, fninkey.end$ executes just as the INKEY$ function would, but adds the ability to recognize the logical end of file character.

```
def fninkey.end$(x,y)
temp$ = inkey$(x,y)
when index(temp$,chr$(16R1A)) <> 0
  seterror 2 : ' force an end file error
whend
fnend$ = temp$
```

INKEY$(x,y) may be used only with sequential files. An attempt to access a random or unfmt file will generate a wrong file type error.

see also: INCHAR$ (section 4.086) and file handling (section 6)

4.091 INP

Format: INP(x)

Type: intrinsic function

Example: A = INP(X)

The INP function returns a numeric value, in the range of 0 through 255, which is equivalent to the value of a byte read from i/o port number x. The numeric expression x must evaluate in the range of 0 through 255.

4.092 INPUT

Format: INPUT [slit;] var {,var} [;]

Type: statement

Purpose: to allow console input

Example:
```
INPUT A,B
INPUT "Enter data"; A$
```

The INPUT statement is one of several methods provided by BASIC/Z to allow console input. Upon execution, the string literal is displayed as an optional prompt, followed by a question mark (?).

It is expected that the user will type a line of data, which is terminated by pressing the <RETURN> key. During entry, the standard back-space and cancel entire line functions are recognized, as defined at installation. All other non-printing characters are ignored. A special case occurs when the user enters a null line by pressing just <RETURN>. In this instance, all variable arguments will be left with their values intact, and execution of the program will continue at the next statement. Depending upon the program logic, it may be wise to initialize variables before an INPUT statement is executed.

The data that is typed is assigned to the variable(s) specified, and must match in type (numeric or string). If more than one data item is entered, they must be separated by the current string delimiter. If too few items are typed for the variable list, an insufficient data error will be generated.

If a trailing semicolon is included, then the terminating <RETURN> typed is not echoed, to allow for explicit cursor positioning.

see also: EDIT$ (section 4.047)

4.093 INPUT$

Format: INPUT$(x)

Type: intrinsic function

Example: TEXT$ = INPUT$(20)

INPUT$ is functionally identical to the simplified version of EDIT$, which is described in section 4.047. It is recommended that all occurrences of INPUT$(x) in existing programs be changed to EDIT$(x), as it is likely that this function will be eliminated from future revisions of BASIC/Z.

4.094 INSTAT

Format: INSTAT

Type: intrinsic function

Example: IF INSTAT THEN TEXT$ = EDIT$(12)

The INSTAT function is used to test the console status, without actually returning an input character, if any. If a key has been pressed, it returns a logical true (1). If not, it returns a logical false (0).

In the above example, an input character which sets INSTAT true is not cleared until execution of EDIT$, which accepts it as the first keystroke.

4.095 INT

Format: INT(x)

Type: intrinsic function

Example: A = INT(X)

The function INT returns a numeric value, which is the greatest integer smaller or equal to the numeric expression x.

The difference between the INT and FIX functions is that INT returns the next lower integer for a negative value (i.e. FIX(-4.5) = -4 while INT(-4.5) = -5).

4.096 LABEL

Format: [LABEL] @label

Type: statement

Purpose: to create a symbolic destination for GOTO, GOSUB, etc.

Example: LABEL @PRINT.ROUTINE

LABEL is used to designate a symbolic destination for GOTO, GOSUB, and RESTORE statements. As such, the symbolic name may be used in place of a line number reference, for greater program clarity. When a label is referenced, as in GOSUB @XXX, control is transferred to the first executable statement following the label. In the example below, the reference gosub @print.it in line 100 actually transfers control to line 140.

```
100 gosub @print.it
110 end
120 '
130 @print.it
140 print text$; text.end$;
150 return
```

The extensive use of labels in your programs can do much to aid in documenting them, as they tend to illustrate program flow. Labels add absolutely no overhead to your running programs, so long, meaningful names are appropriate!

A label name must begin with '@' and a letter, followed by any combination of letters, numbers, and periods. In designating a symbolic location, the word LABEL is optional, and is generally omitted.

4.097 LASTFILE

Format: LASTFILE

Type: intrinsic function

Example: IF LASTFILE = 14 THEN GOTO @ERR.ROUTINE.4

The function LASTFILE returns a numeric value, in the range of 0 through 29, equivalent to the file number of the most recent disk access, whether or not successful. If the most recent access generated an invalid file number error, then the value 30 is returned.

It is probable that LASTFILE will be used most frequently in global error trap routines, to determine the source of a disk error.

4.098 LCASE$

Format: LCASE$(x$)

Type: intrinsic function

Example: TEXT$ = LCASE$(TEXT$)

The function LCASE$ returns the argument expression x$, but with every upper case letter converted to the lower case equivalent. All other characters remain unchanged.

see also: UCASE$ (section 4.211)

4.099 LEFT$

Format: LEFT$(x$,x)

Type: intrinsic function

Example: A$ = LEFT$(X$,X)

The function LEFT$ returns a string comprised of the leftmost x characters of the string expression x$. The numeric expression x must evaluate in the range of 0 through 250. If x is greater than LEN(x$), the entire string x$ is returned. If x equals zero (0), a null string is returned.

see also: MID$ (section 4.115) and RIGHT$ (section 4.176)

4.100 LEN

Format: LEN(x$)

Type: intrinsic function

Example: A = LEN(X$)

The function LEN returns a numeric value, in the range of 0 through 250, which is equivalent to the number of characters in x$. All characters are counted, including any which may be non-printable.

4.101 LET

Format: [LET] var = expression

Type: statement

Purpose: to assign the value of an expression to a variable

Example:
```
LET A$="Hello"
A=123
```

The LET statement is used to assign a value to a variable. The variable, shown to the left of the equal sign, may be either scalar or array.

Upon execution, the expression to the right of the equal sign is evaluated, and assigned to the designated variable. The expression may yield either a string or a numeric result, but it must match in type to the variable. The word LET is optional, and is generally omitted.

4.102 LINK

Format: LINK x$ [,y$]

Type: statement

Purpose: to load and execute a command file (type .COM)

Example: LINK "BZ", "PROGRAM"

The LINK statement will allow you to load and execute a command file. Upon execution, the command file named by x$ is loaded into the TPA (transient program area), and executed at the TBASE address. As the command file is assumed to have an extension of .COM, it need not be included in x$. If a drive name is not included, the currently selected default drive is assumed.

The second argument y$ is optional. If present, it will be passed to the command file as a command line trailer.

4.103 LOAD

Format: LOAD x$, x

Type: statement

Purpose: to load an object file into memory

Example: LOAD "PROGRAM3", ENDMEM-16R1000

The LOAD statement is used to read an object file into memory, for access by user-defined, assembly language functions. Absolutely no interpretation or translation is performed on the file as it is loaded. It is assumed to be an absolute memory image (just as a .COM file), which is "ready-to-run" at the designated load address. This type of memory image file may be created through the use of the DDT utility, supplied with your operating system.

Upon execution, the object file named by x$ is loaded into memory at the address specified by the numeric expression x. If x$ does not include a drive name, the currently selected default drive will be assumed.

Prior to execution of LOAD, a MEMEND statement must be executed to reserve space for the object file. Since the entire file will be read into memory (in full, 128 byte records), caution must be exercised to insure that sufficient space is reserved, to avoid overwriting any needed program or data.

4.104 LOCK

Format: LOCK(x,y)

Type: intrinsic function

Example: IF NOT LOCK(1,42) THEN GOTO @WAIT.FOR.UNLOCK

The LOCK function is used to lock record number y in file number x. A record which is locked is one which may not be written to nor locked by any other process. This function is used to prevent "interleaved updating", whereby two programs update the same data simultaneously, causing one of the updates to be overwritten, and lost. Generally speaking, a data record should be locked prior to any reference (even reading), if there is any possibility it may be updated, and unlocked only when all possible updates are completed.

The LOCK function return a logical true (1) if the requested operation was completed successfully, or false (0) if this record is presently locked by another process. If your operating system does not support record locking, every LOCK request with valid arguments is considered to be successful.

see also: UNLOCK (section 4.214)

4.105 LOCKED

Format: LOCKED

Type: option

Example: OPEN 1 "FILE" LOCKED

LOCKED is an option which may be designated as part of an OPEN or CREATE statement. The reserved word LOCKED is optional, as this is the default mode for a file which is neither UNLOCKED nor READONLY.

A LOCKED file is one which may not be written to, locked, renamed, etc., by any other process until it is released with a CLOSE statement.

see also: UNLOCKED (section 4.215) and READONLY (section 4.163)

4.106 LOG

Format: LOG(x)

Type: intrinsic function

Example: A = LOG(X)

The function LOG returns the natural logarithm (base e = 2.71828 ...) of the numeric expression x.

4.107 LOG10

Format: LOG10(x)

Type: intrinsic function

Example: A = LOG10(X)

The function LOG10 returns the common logarithm (base 10) of the numeric expression x.

4.108 LOWCASE

Format: LOWCASE

Type: statement

Purpose: to allow lower case console input

Example: LOWCASE

BASIC/Z provides a software "cap-lock" option for executing programs. If this feature is enabled, through execution of an UPCASE statement, all alphabetic console input is forced to upper case, as it is typed. Execution of the statement LOWCASE allows, but does not force, lower case alphabetic input. LOWCASE mode is the default condition, and need not be explicitly initialized.

see also: UPCASE (section 4.217)

4.109 LOWINT

Format: LOWINT

Type: statement

Purpose: to cause low intensity console output

Example: LOWINT

Execution of the LOWINT statement causes subsequent printed output, directed to the console, to be low (dim) intensity, assuming this feature is supported by the console hardware.

see also: HIGHINT (section 4.084), and STDINT (section 4.199).

4.110 LPRINTER

Format: LPRINTER

Type: statement

Purpose: to direct printed output to the printer

Example: LPRINTER

The LPRINTER statement is used to designate printed output to the printer. After execution of LPRINTER, all subsequent output to the print stream (primarily PRINT statements) is directed to the printer only.

see also: CONSOLE (section 4.024), ECHO (section 4.046), NULL (section 4.123), and SPOOL (section 4.196)

4.111 MAX

Format: MAX(x,y)

Type: intrinsic function

Example: A = MAX(X,Y)

The function MAX compares the values of the numeric expressions x and y, and returns the greater of the two.

4.112 MAX$

Format: MAX$(x$,y$)

Type: intrinsic function

Example: A$ = MAX$(X$,Y$)

The function MAX$ compares the values of the string expressions x$ and y$, and returns the greater of the two.

4.113 MEMEND

Format: MEMEND x

Type: statement

Purpose: to define the upper limit of memory usable by BASIC/Z

Example: MEMEND 16RBFFF

The **MEMEND** statement is used to define the upper limit of memory which may be used by the program. The primary need for this function is in allocation of memory space for assembly language subroutines, which must be loaded above the compiled program. The numeric expression x defines the highest address usable by BASIC/Z, and must evaluate in the range of 0 through 65535.

Generally, a **MEMEND** must be executed before an object file is loaded, and that file must be placed in memory above the designated **MEMEND**, yet below the true end of memory. The actual end of memory may be accessed with the **ENDMEM** function. Caution must be exercised to avoid any conflict with the compiled program (below MEMEND) and the operating system (above ENDMEM). A convenient method of reserving 1000 hex bytes might be:

```
100  memend  endmem-16R1000
```

MEMEND statements are executed dynamically, and may be repeated one or more times, to alter the memory end repeatedly. However, all files must be closed at the time of execution. Any file which remains open will be implicitly closed (i.e. disassociated from the file number, but not updated to disk).

A **MEMEND** is not reset by execution of a **CHAIN** or **RUN** statement, so it need not be repeated in a program segment receiving control.

see also: **ENDMEM** (section 4.051) and **LOAD** (section 4.103)

4.114 MID$

Format: MID$(svar,x) = x$

Type: statement

Purpose: to replace part of a string variable with an expression

Example: MID$(A$,2) = B$

Upon execution of a MID$ statement, the string expression x$ will replace characters in the designated string variable, starting at position x. Generally, the number of characters replaced will be equal to the length of the expression x$. However, replacement of characters will never continue beyond the original length of the string variable.

see also: MID$ function (section 4.115)

4.115 MID$

Format: MID$(x$,x,y)

Type: intrinsic function

Example: A$ = MID$(B$,C,D)

The function MID$ returns a string comprised of y characters extracted from the string expression x$, starting with the character at position x. The numeric expression x must evaluate in the range of 1 through 250, while y may range from 0 through 250.

If y equals zero (0) or x points to a position past the end of the string, a null string is returned. If y is greater than the number of characters from position x through the end of the string, then all characters from position x through the end of the string are returned.

see also: MID$ statement (section 4.114)

4.116 MIN

Format: MIN(x,y)

Type: intrinsic function

Example: A = MIN(X,Y)

The function MIN compares the values of the numeric expresions x and y, and returns the lesser of the two.

4.117 MIN$

Format: MIN$(x$,y$)

Type: intrinsic function

Example: A$ = MIN$(X$,Y$)

The function MIN$ compares the value of the string expressions x$ and y$, and returns the lesser of the two.

4.118 MOD

Format: MOD(x,y)

Type: intrinsic function

Example: A = MOD(X,Y)

The function MOD returns a numeric value of x modulo y. This is equivalent to x-(y*int(x/y)).

4.119 NAME

Format: NAME(x)

Type: intrinsic function

Example: A$ = NAME(1)

The function NAME returns an eleven byte string equivalent to the name of the disk file associated with file number x. The numeric expression x must evaluate in the range of 0 through 29.

4.120 NEXT

Format: NEXT nvar

Type: statement

Purpose: to terminate the looping structure of a FOR statement

Example: NEXT INDEX%

The NEXT statement is used to terminate a FOR/NEXT loop, in which both reference the same numeric counter variable. A complete definition of this looping structure may be found under FOR (section 4.070).

4.121 NOBLINK

Format: NOBLINK

Type: statement

Purpose: to cause console output to not flash

Example: NOBLINK

Execution of the NOBLINK statement causes subsequent printed output, directed to the console, to not flash. NOBLINK mode is the default condition, and need not be explicitly initialized.

see also: BLINK (section 4.010)

4.122 NONLINE

Format: NONLINE

Type: statement

Purpose: to cause console output to not be underlined

Example: NONLINE

Execution of the NONLINE statement causes subsequent printed output, directed to the console, to not be underlined. NONLINE mode is the default condition, and need not be explicitly initialized.

see also: ULINE (section 4.212)

4.123 NULL

Format: NULL

Type: statement

Purpose: to suppress printed output

Example: NULL

The NULL statement is used to suppress all printed output. After execution of NULL, all subsequent output to the print stream (primarily PRINT statements), is discarded.

see also: CONSOLE (section 4.024), ECHO (section 4.046), LPRINTER (section 4.110), and SPOOL (section 4.196)

4.124 NVIDEO

Format: NVIDEO

Type: statement

Purpose: to cause console output to be normal video

Example: NVIDEO

Execution of the NVIDEO statement causes subsequent printed output, directed to the console, to be normal video. NVIDEO mode is the default condition, and need not be explicitly initialized.

see also: RVIDEO (section 4.179)

4.125 OCT$

Format: OCT$(x)

Type: intrinsic function

Example: X$ = OCT$(X#)

The OCT$ function returns a six character string which contains the split octal representation of the argument x. The numeric expression x must evaluate in the range of 0 through 65535.

4.126 ON END x CLEAR

Format: ON END x CLEAR

Type: statement

Purpose: to disable an end file trap

Example: ON END 24 CLEAR

BASIC/Z allows each open file to maintain an end file trap, which may be declared with an END option (section 4.050), or an ON END x GOTO statement (section 4.127).

Under certain conditions, it may be desirable to have that trap enabled for just a limited time. Upon execution of this statement, an end file trap previously set for file number x will be disabled. An end of file condition will then be treated as any other error.

see also: error handling (section 7)

4.127 ON END x GOTO

Format: ON END x GOTO dest

Type: statement

Purpose: to enable an end file trap

Example: ON END 11 GOTO @END.FILE.TRAP

BASIC/Z allows each open file to maintain an end file trap (i.e. a destination to automatically GOTO upon an attempt to read past the end of file). The argument dest must evaluate to a label or line number at the end file routine.

An end file trap may be declared with ON END x GOTO, or with an END option (section 4.050). It may be explicitly disabled with an ON END x CLEAR statement, or implicitly disabled upon transfer of control to another program segment, using a CHAIN or RUN statement. The ON END x GOTO statement may be used repeatedly, to alter the destination address of an end file trap.

see also: error handling (section 7)

4.128 ON ERROR x CLEAR

Format: ON ERROR x CLEAR

Type: statement

Purpose: to disable a local error trap

Example: ON ERROR 24 CLEAR

BASIC/Z allows each open file to maintain a local error trap, to handle just disk errors specific to that file. A local error trap may be declared with an ERROR option (section 4.060), or an ON ERROR x GOTO statement (section 4.129).

Under certain conditions, it may be desirable to have that trap enabled for just a limited time. Upon execution of this statement, a local error trap previously set for file number x will be disabled. Any subsequent disk errors relating to file number x would then be handled by a global error trap, if one has been enabled.

see also: error handling (section 7)

4.129 ON ERROR x GOTO

Format: ON ERROR x GOTO dest

Type: statement

Purpose: to enable a local error trap

Example: ON ERROR 11 GOTO @LOCAL.TRAP

BASIC/Z allows each open file to maintain a local error trap, to handle just disk errors specific to that file. The argument dest must evaluate to a label or line number at the local error routine.

A local error trap may be declared with an ON ERROR x GOTO statement, or with an ERROR option (section 4.060). It may be explicitly disabled with an ON ERROR x CLEAR statement, or implicitly disabled upon transfer of control to another program segment, using a CHAIN or RUN statement. The ON ERROR x GOTO statement may be used repeatedly, to alter the destination address of a local error trap.

see also: error handling (section 7)

4.130 ON ERROR CLEAR

Format: ON ERROR CLEAR

Type: statement

Purpose: to disable global error trapping

Example: ON ERROR CLEAR

BASIC/Z supports the concept of global error trapping, which is enabled with the ON ERROR GOTO statement (section 4.131).

Under certain conditions, it may be desirable to disable global error trapping, such as within the error handling routine itself. Failure to do so could cause a misleading endless loop within that routine:

```
100 @error.handler
110 close
120 print err$; " with file number"; lastfile
130 goto @terminate.program
```

Upon first glance, the above error routine would appear to operate properly. However, it is important to consider the nature of the error which caused the branch to this routine originally. If, for example, a file could not be accessed because a floppy drive was not loaded, then the close statement in line 110 will generate yet another error! An endless loop would be executed from line 100, 110, 100, 110, etc., while the system would appear to be totally dead.

To avoid this problem, ON ERROR CLEAR may be executed at the start of the routine, to disable error trapping. Any subsequent errors will then cause a descriptive message to be printed, and program execution will be terminated.

see also: error handling (section 7)

4.131 ON ERROR GOTO

Format: ON ERROR GOTO dest

Type: statement

Purpose: to enable global error trapping, and to specify the label or line number of the error handling routine

Example: ON ERROR GOTO @ERR.HANDLER

BASIC/Z supports two levels of error trapping, termed local and global. Each open file may maintain an individual local error trap, to handle just disk errors specific to that file. Generation of any error not controlled by a local error trap will cause control to be transferred to the destination label or line number specified by the most recently executed ON ERROR GOTO statement. Global error trapping may be disabled with the ON ERROR CLEAR statement (section 4.130).

When an error occurs, the error type may be identified with the functions ERR and ERR$. A full definition of all error codes will be found in appendix C of this manual.

If global error trapping is not enabled, generation of an error will cause a descriptive message to be printed, and program execution will be terminated.

see also: error handling (section 7)

4.132 ON x GOSUB

Format: ON x GOSUB dest {,dest}

Type: statement

Purpose: to conditionally execute one of several subroutines

Example: ON A+1 GOSUB 500,600,700,800

The ON x GOSUB statement causes execution of the subroutine beginning at the label or line number whose positional value in the destination list is equal to the value of the expression (i.e. if x=1, the first would be executed, i x=2, the second, etc.). If the value of the expression is less than one, or greater than the number of items in the destination list, no subroutine is executed, and the program continues at the next executable statement.

4.133 ON x GOTO

Format: ON x GOTO dest {,dest}

Type: statement

Purpose: to conditionally transfer control to one of several destinations

Example: ON B-3 GOTO 200,210,220,230,240

The ON x GOTO statement causes control to be transferred to the label or line number whose positional value in the destination list is equal to the value of the expression (i.e. if x=1, the first destination, if x=2, the second, etc.). If the value of the expression is less than one, or greater than the number of items in the destination list, the program continues at the next executable statement.

4.134 OPEN

Format: OPEN x x$ [options]

Type: statement

Purpose: to allow disk i/o to a file

Example: OPEN 21 "B:DATAFILE" ERROR @ERR.HANDLER READONLY

A disk file must be opened before any operations can be performed on that file. If the file to be opened does not already exist, the CREATE statement (section 4.026) must be used in place of OPEN.

The numeric expression x (in the range of 0 through 29) specifies a file number, which will be associated with this file until it is closed. The expression x$ must evaluate to a string containing an optional drive name (if other than the current default), and the name of the file to be opened. Options may be chosen, as needed, from the selection below. File lock options (LOCKED, READONLY, UNLOCKED) may not be combined. If none are chosen, a locked file is assumed. File lock options are ignored in a single user system.

OPTIONS :

CLEAR - The CLEAR option is used to create the condition of a logically empty file, by resetting the PUT and EOF pointers to the start of the file.

END - The END option is used to designate a destination to GOTO upon an attempt to read past the logical end of this file. It must be immediately followed by a valid line number or label.

ERROR - The ERROR option is used to designate a destination to GOTO if a disk error occurs in a reference to this file. It must be immediately followed by a valid line number or label.

LOCKED - Writing to this file, by other users, is prohibited. If a file is not specified as READONLY or UNLOCKED, then a default to LOCKED will be assumed.

READONLY - The file may be shared, on a read/only basis, with other users.

continued - -

4.134 OPEN (continued)

RECLEN - If RECLEN is not included, the file is assumed to be sequential type. If present, it must be immediately followed by a numeric expression which evaluates to the desired logical record length for a random or unfmt file. This must be greater than zero, and smaller or equal to 250 or the reclen limit specified in a SIZES statement. The RECLEN of a random file may never be changed from the explicit value declared when it was created. With a random (but not unfmt) file, an implied form of this option is recognized. In this form, a RECLEN of 0 may be specified, and BASIC/Z will supply the correct value. The RECLEN of a file may be retrieved with the RECSIZE function (section 4.168).

UNFMT - UNFMT specifies a simplified form of random access file (a RECLEN option is mandatory) which eliminates the header record. This allows compatibility with other languages, but may introduce certain errors in the PUT and EOF pointers when the logical record length is set to less than 128. Also, in the UNFMT mode, the logical record length and logical file size (if altered with the EOF statement) are not maintained by the system, but just discarded when the file is closed. This precludes use of an implied RECLEN with this file type.

UNLOCKED - The file may be shared, on a read/write basis, with other users.

see also: CREATE (section 4.026) and file handling (section 6)

4.135 OUT

Format: OUT(x) = y

Type: statement

Purpose: to output a byte to a port

Example: OUT(6) = 30

Execution of the OUT statement will cause the value of numeric expression y to be output to the i/o port which is specified by numeric expression x. Both of the expressions must evaluate in the range of 0 through 255.

4.136 PAGESIZE

Format: PAGESIZE x, y

Type: statement

Purpose: to allow program control of printer page size

Example: PAGESIZE 88, 132

Upon initial execution of a compiled program, a default printer page size of 66 lines by 132 columns is assumed. The PAGESIZE statement may be used to alter these values under program control.

Numeric expression x specifies the logical number of lines per page, and numeric expression y the logical columns per line.

At the time of execution, BASIC/Z assumes that the printer is set to "top-of-form". The maximum possible value for either parameter is 255.

4.137 PCOL

Format: PCOL

Type: intrinsic function

Example: IF PCOL > 60 THEN PRINT

The PCOL function returns a numeric value equal to the current column position of the cursor on the printer.

4.138 PEEK

Format: PEEK(x)

Type: intrinsic function

Example: A% = PEEK(X%)

The PEEK function returns a numeric value equal to the contents of memory location x. The value returned will be an integer in the range of 0 to 255. The expression x must evaluate in the range of 0 through 65535.

4.139 PEEKWORD

Format: PEEKWORD(x)

Type: intrinsic function

Example: A% = PEEKWORD(X%)

The PEEKWORD function returns a numeric value equal to the contents of memory location x and x+1, which are assumed to be in low/high format. The value returned will be an integer in the range of 0 to 65535. The expression x must evaluate in the range of 0 through 65535.

4.140 PHIGH

Format: PHIGH

Type: statement

Purpose: to print double height characters on the printer

Example: PHIGH

Execution of the PHIGH statement causes subsequent printed output, directed to the printer, to be double height characters, assuming that this feature is supported by the printer hardware.

see also: PLOW (section 4.143)

4.141 PI

Format: PI

Type: intrinsic function

Example: PRINT PI * 2

The function PI returns the numeric value of pi (3.14159...), rounded off to the precision implied by RSIZE.

4.142 PLINE

Format: PLINE

Type: intrinsic function

Example: IF PLINE > 58 THEN FORMFEED

The PLINE function returns a numeric value equal to the current line position of the cursor on the printer.

4.143 PLOW

Format: PLOW

Type: statement

Purpose: to print single height characters on the printer

Example: PLOW

Execution of the PLOW statement causes subsequent printed output, directed to the printer, to be single height characters. PLOW mode is the default condition, and need not be explicitly initialized.

see also: PHIGH (section 4.140)

4.144 PNARROW

Format: PNARROW

Type: statement

Purpose: to print single width characters on the printer

Example: PNARROW

Execution of the PNARROW statement causes subsequent printed output, directed to the printer, to be single width characters. PNARROW mode is the default condition, and need not be explicitly initialized.

see also: PWIDE (section 4.159)

4.145 POKE

Format: POKE(x) = y

Type: statement

Purpose: to store a byte in memory

Example: POKE(16R900F) = 30

Execution of the POKE statement causes the value specified by y to be stored at memory location x. The numeric expression x must evaluate in the range of 0 through 65535, while y must evaluate in the range of 0 through 255.

see also: PEEK (section 4.138)

4.146 POKEWORD

Format: POKEWORD(x) = y

Type: statement

Purpose: to store two bytes (low/high) in memory

Example: POKEWORD(16R900F) = 16RFFFF

Execution of the POKEWORD statement causes the value y to be stored, in low/high format, at memory locations x and x+1. The numeric expressions x and y must each evaluate in the range of 0 through 65535.

4.147 POP

Format: POP

Type: statement

Purpose: to delete an address from the subroutine stack

Example: POP

Execution of the statement POP will cause the most recent return address to be deleted from the subroutine stack. If the subroutine stack is empty, an "invalid return" error is generated.

see also: CLRSUB (section 4.020) and PUSH (section 4.150)

4.148 PRINT

Format: PRINT [expression list]

Type: statement

Purpose: to output data to the print stream

Example: PRINT A,B,C,D

The PRINT statement is used to output data to the print stream, for display on the console or printer, or for spooling to a disk file. Upon program execution, this stream is directed to the console by default. However, re-direction statements (ECHO, LPRINTER, NULL, and SPOOL) may be executed to direct the stream to one or more other devices.

Expressions in the expression list may be either string or numeric. Each expression is evaluated individually, and truncated if it exceeds 250 characters in length. String expressions are output to the print stream without change. However, non-printing characters should be avoided, as errors may be introduced in the cursor positioning functions. To output control code sequences, use the PRINTER statement (section 4.149) or VIDEO statement (section 4.222) instead. Numeric expressions are converted to strings, following the rules stated in Numeric output formats (section 2.024), before output to the print stream.

If the PRINT statement appears alone, a carriage return/line feed combination is output to the print stream. If expressions follow it, they are evaluated and output to the stream, generally followed by a carriage return/line feed combination.

Each expression in the expression list must be delimited from the next by either a comma (,) or a semicolon (;). However, if the final expression in the expression list is followed by either one, then output from subsequent PRINT statements will begin at this position (i.e. the trailing carriage return/line feed is suppressed).

If an expression is followed by a semicolon, the print position is left immediately following the last character output.

If an expression is followed by a comma, the print position is advanced to the beginning of the next 16 character print field, to allow easy columnar placement of data.

The print position may be set explicitly by including a TAB function. For more information, please refer to TAB (section 4.206).

4.149 PRINTER

Format: PRINTER(x)

Type: statement

Purpose: to output a user-defined series of control codes to the printer

Example: PRINTER(7)

The PRINTER statement may be used to control font size or other printer logic under program control. Upon execution, a user-defined series of control codes is output to the printer. This is generally preferable to use of a PRINT statement, to avoid hardware dependence, and to avoid the possible introduction of errors in the PCOL and PLINE functions.

Up to eight (8) sets of control codes may be defined with the INSTALL utility program. Depending upon the value of the expression x (which must range between 0 and 7), the corresponding set of control codes will be output.

see also: VIDEO (section 4.222)

4.150 PUSH

Format: PUSH dest

Type: statement

Purpose: to add an address to the subroutine stack

Example: PUSH 100

Execution of the PUSH statement will cause the address of the line number or label to be added to the top of the subroutine stack. The next RETURN executed would cause control transfer to the statement at that destination.

```
push @routine
return
```

Although the above example would be inefficient, it would produce precisely the same result as if GOTO @routine were executed.

see also: POP (section 4.147)

4.151 PUT

Format: PUT x [RECORD y] expression list

Type: statement

Purpose: to write a record to a random or unfmt disk file

Example: PUT 1 RECORD 4 D; E$+"," ; F

The PUT statement is used to write one logical record to the random or unfmt file associated with file number x. If the RECORD option is included, the record written will be that specified by the numeric expression y. If the RECORD option is not included, the record written will be that specified by the indexed PUT pointer, which will then be incremented, after the write.

The logical record to be written is composed in a buffer termed the record string. The length of this record string is determined by the RECLEN option when the file was opened or created. If the data to be written does not fill the record string, it will be automatically padded will nulls (binary 0). Any data written in excess of the record string length will be truncated.

All four random disk write statements (PUT, PUTNUM, PUTVEC, and PUTVECSPC) access the file and write a logical record in the same manner. The difference between them is the nature of the data they deal with, and the format of the data within the record which is constructed.

The PUT statement constructs a record string much like a print line. The only exceptions are that TAB functions are illegal, and that a carriage return/line feed is not appended. The expression list may consist of one or more expressions, which may be either string or numeric. Each expression is evaluated individually, and truncated if it exceeds 250 characters in length. String expressions are inserted into the record string without change. Numeric expressions are converted to strings, following the rules stated in Numeric output formats (section 2.024), before inserting them into the record string.

Each expression in the list must be delimited from the next by either a semicolon (;) or a comma (,). However, these act as punctuation only, and are not inserted into the record string. The final position of each data item in the record string is determined both by the preceding data, and by the punctuation used to delimit the items in the expression list. The record string may be considered to be partitioned into zones of 16 characters each. If an expression is followed by a comma (,) the record string is padded with enough spaces so that the next expression will be positioned at the beginning of a 16 character partition. If an expression is followed by a semicolon (;) the record string is not padded at all. The next expression will be appended at the next position.

continued - -

4.151 PUT (continued)

The PUT statement may be used to write any of the 256 possible codes as a character in the record string. As described previously, it may pad numerics and expressions delimited by a comma with blanks, but it never, under any circumstances, inserts any other delimiters between the expressions. If you wish to utilize the concept of variable length sub-fields, the current string delimiter must be explicitly included in the expression list. For example:

```
100 delim$ = ","
110 string delim$
120 put 1 record 1 a$; delim$; b$; delim$; c$; delim$
130 get 1 record 1 x$, y$, z$
```

Upon execution of the above, x$, y$, and z$ would be set precisely equal to a$, b$, and c$ respectively.

Special care must be taken whenever a record string is constructed which is shorter than the logical record length for that file. Generally speaking, the current string delimiter should be appended to insure that a GET used to retrieve the data will be able to ascertain the correct length.

PUT may only be used with random or unfmt files. An attempt to write to a sequential file will generate a wrong file type error.

see also: GET (section 4.074) and file handling (section 6)

4.052 PUT$

Format: PUT$ x expression list

Type: statement

Purpose: to write data to a sequential file

Example: PUT$ 1 X$; Y$; Z$

The PUT$ statement is used to write data to the sequential file associated with file number x. Unlike the other sequential disk write statements (PUTFIELD and PUTSEQ), no field delimiters nor record delimiters are appended to the data. This allows writing a partial field or partial logical record, to which additional data may be appended. However, caution must be exercised, as data which remains undelimited will not be properly recognized by the GETFIELD or GETSEQ statements.

The data to be written is composed in a buffer termed the record string. The PUT$ statement constructs a record string much like a print line. The only exceptions are that TAB functions are illegal, and that a carriage return/line feed is not appended. The expression list may consist of one or more expressions, which may be either string or numeric. Each expression is evaluated individually, and truncated if it exceeds 250 characters in length. String expressions are inserted into the record string without change. Numeric expressions are converted to strings, following the rules stated in Numeric output formats (section 2.024), before inserting them into the record string.

Each expression in the list must be delimited from the next by either a semicolon (;) or a comma (,). However, these act as punctuation only, and are not inserted into the record string. The final position of each data item in the record string is determined both by the preceding data, and by the punctuation used to delimit the items in the expression list. The record string may be considered to be partitioned into zones of 16 characters each. If an expression is followed by a comma (,) the record string is padded with enough spaces so that the next expression will be positioned at the beginning of a 16 character partition. If an expression is followed by a semicolon (;) the record string is not padded at all. The next expression will be appended at the next position.

The PUT$ statement may be used to write any of the 256 possible codes as a character in the record string. As described previously, it may pad numerics and expressions delimited by a comma with blanks, but it never, under any circumstances, inserts any other delimiters between the expressions.

PUT$ may only be used with sequential files. An attempt to write to a random or unfmt file will generate a wrong file type error.

see also: file handling (section 6)

4.153 PUTFIELD

Format: PUTFIELD x expression list

Type: statement

Purpose: to write a field to a sequential file

Example: PUTFIELD 1 X$; Y$; Z$

The PUTFIELD statement is used to write one logical field to the sequential file associated with file number x. It is functionally identical to the PUT$ statement (section 4.152), with the single exception that the current string delimiter is automatically appended to the data. This allows it to be easily read with the GETFIELD statement (section 4.075).

```
100 string ","
110 put$  1  x$; ","
120 putfield  1  x$
```

In the above example, the statements in line 110 and line 120 would write precisely the same data to the file.

PUTFIELD may only be used with sequential files. An attempt to write to a random or unfmt file will generate a wrong file type error.

see also: file handling (section 6)

4.154 PUTNUM

Format: PUTNUM x [RECORD y] nvar {,nvar}

Type: statement

Purpose: to write numeric data to a random or unfmt disk file

Example: PUTNUM 17 A, B#, C&, D%, E&(11), E&(12)

The PUTNUM statement is used to write one logical record to the random or unfmt file associated with file number x. If the RECORD option is included, the record written will be that specified by the numeric expression y. If the RECORD option is not included, the record written will be that specified by the indexed PUT pointer, which will then be incremented, after the write.

continued - -

4.154 PUTNUM (continued)

The logical record to be written is composed in a buffer termed the record string. The length of this record string is determined by the RECLEN option when the file was opened or created. If the data to be written does not fill the record string, it will be automatically padded will nulls (binary 0). Any data written in excess of the record string length will be truncated.

All four random disk write statements (PUT, PUTNUM, PUTVEC, and PUTVECSPC) access the file and write a logical record in the same manner. The difference between them is the nature of the data they deal with, and the format of the data within the record which is constructed.

The PUTNUM statement constructs a record string of numeric data in the internal binary or bcd format described in section 2.010 through 2.013. No delimiters of any type are inserted between data items. Rather, numeric data items are packed immediately adjacent to the next. It is assumed that, when retrieved, they will be parsed based upon the known size of each.

```
100 sizes(6,4,250,250)
110 putnum 1 record 1 x,y,z#
```

```
100 sizes(6,4,250,250)
110 rec$ = cvtr$(x)
120 rec$ = rec$ + cvtr$(y)
130 rec$ = rec$ + cvtcd$(z#)
140 put 1 record 1 rec$
```

The above two examples provide the identical function, although the first is substantially more efficient. They each create a record consisting of 14 bytes of significant numeric data in the specified internal formats.

Generally speaking, a record written with PUTNUM is read back with the GETNUM statement (section 4.076). Although data types may be intermixed in a record, extreme care must be used to insure that a value is written and read from the same variable type, so that the same format, and number of bytes, will be expected. Also, in the case of integer and real variables, it is mandatory that ISIZE and RSIZE be identical at the time of reading and writing.

PUTNUM may only be used with random or unfmt files. An attempt to write to a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.155 PUTSEEK

Format: PUTSEEK(x) = y

Type: statement

Purpose: to explicitly set the indexed PUT pointer

Example: PUTSEEK(1) = 50

In writing to a random or unfmt disk file with PUT, PUTNUM, PUTVEC, or PUTVECSPC, the record number may be stated explicitly with the RECORD option, or it may be implied as the current value of the indexed PUT pointer. A separate PUT pointer is automatically maintained for each file. Each time that a file is opened, the PUT pointer is initialized to point to the record following the last record in the file. Every subsequent write without a record option utilizes the current value, and then increments the pointer by one (1). This allows sequential access to a random or unfmt file with minimum difficulties.

The PUTSEEK statement may be used to reset the PUT pointer for file number x to the record specified by the numeric expression y. It must evaluate to a number greater than 0, and less than 65536.

The current value of the indexed PUT pointer may be accessed with the RECPUT function (section 4.167).

PUTSEEK may only be used with random or unfmt files. An attempt to reference a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.156 PUTSEQ

Format: PUTSEQ x expression list

Type: statement

Purpose: to write a logical record to a sequential file

Example: PUTSEQ 1 X$; Y$; Z$

The PUTSEQ statement is used to write one logical record to the sequential file associated with file number x. It is functionally identical to the PUT$ statement (section 4.152), with the single exception that the record delimiter (carriage return/line feed combination) is automatically appended to the data. This allows it to be easily read with the GETSEQ statement (section 4.078).

```
100 put$  1  x$; chr$(13); chr$(10)
110 putseq  1  x$
```

In the above example, the statements in line 100 and line 110 would write precisely the same data to the file.

PUTSEQ may only be used with sequential files. An attempt to write to a random or unfmt file will generate a wrong file type error.

see also: file handling (section 6)

4.157 PUTVEC

Format: `PUTVEC x [RECORD y] navar, navar`
`PUTVEC x [RECORD y] savar, savar, x$`

Type: statement

Purpose: to write data to a random or unfmt disk file from successive array elements

Example:
```
PUTVEC 1 A#(0), A#(99)
PUTVEC 1 A$(4), A$(23), MASK$
```

The PUTVEC statement was designed to allow a single statement to write a large number of data items, from successive array elements, to a logical record of random or unfmt disk file number x. If the RECORD option is included, the record written will be that specified by the numeric expression y. If the RECORD option is not included, the record written will be that specified by the indexed PUT pointer, which will then be incremented, after the write.

The logical record to be written is composed in a buffer termed the record string. The length of this record string is determined by the RECLEN option when the file was opened or created. If the data to be written does not fill the record string, it will be automatically padded will nulls (binary 0). Any data written in excess of the record string length will be truncated.

All four random disk write statements (PUT, PUTNUM, PUTVEC, and PUTVECSPC) access the file and write a logical record in the same manner. The difference between them is the nature of the data they deal with, and the format of the data within the record which is constructed.

The two array variable arguments specify the first and the last element of the array to be written, while all intervening elements are presumed. They must reference elements of the same array, with the first preceding the second in memory sequence. PUTVEC is most frequently used with arrays which are one dimensional. However, with care, it may be used with multi-dimensional arrays as well. As described under Array Variables (section 2.016), arrays are stored in column major order, which must be carefully considered. For example:

```
1) putvec 1 x(0), x(9)
2) putvec 1 a(0,0), a(3,2)
3) putvec 1 a(3,0), a(0,1)
4) putvec 1 a(0,2), a(2,0)
```

Example 1 references 10 consecutive elements in a one dimensional array. The remaining examples all reference a two dimensional array which was dimensioned as a(3,2). Example 2 references all 12 elements

continued - -

4.157 PUTVEC (continued)

of the array, in the sequence they are stored in memory. Example 3 references just 2 elements of the array, as they are located adjacent in memory. Example 4 is illegal, as the second precedes the first in memory sequence.

There are two general forms of the PUTVEC statement, which differ depending upon the type of the data, numeric or string.

NUMERIC ARRAY:

The PUTVEC statement constructs a record string of numeric data in the internal binary or bcd format described in section 2.010 through 2.013. No delimiters of any type are inserted between data items. Rather, numeric data items are packed immediately adjacent to the next. It is assumed that, when retrieved, they will be parsed based upon the known size of each. Generally, a record written with this statement will be read with GETVEC (section 4.079), or possibly with GETNUM (section 4.076). However, great care must be exercised to insure that data is read and written from the same variable type, so that the same format, and number of bytes, will be expected. Also, in the case of integer and real variables, it is mandatory that ISIZE and RSIZE be identical at the time of reading and writing.

STRING ARRAY:

This form of the PUTVEC statement constructs a record consisting of string data, in fixed length sub-fields of pre-determined length. No delimiters of any type are inserted between data items in the record, allowing any of the 256 possible characters to be written. Therefore, it is the programmer's responsibility to maintain data on the size and number of sub-fields in a record. In constructing the record string, the number of bytes taken from each array element is based upon the ascii code of the byte of the mask x$ which corresponds to the relative position of the array element. For example:

```
mask$ = chr$(2) + chr$(17) + chr$(8) + chr$(8)
getvec 1 a$(3), a$(6), mask$
```

The above example constructs a record of 35 bytes, the sum of the 4 mask characters. The first 2 bytes of the record are taken from a$(3), the next 17 from a$(4), the next 8 from a$(5), and the final 8 from a$(6). If any array element contains a smaller number of bytes than specified by the mask string, that sub-field will be padded to the designated length with nulls (binary 0). Generally, a record written with this statement is read with GETVEC (section 4.079) or GETVEC$ (section 4.080).

PUTVEC may only be used with random or unfmt files. An attempt to write to a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.158 PUTVECSPC

Format: PUTVECSPC x [RECORD y] savar, savar, x$

Type: statement

Purpose: to write data to a random or unfmt disk file from successive array elements

Example: PUTVECSPC 1 A$(4), A$(23), MASK$

The PUTVECSPC statement is very similar in function to the string form of the PUTVEC statement (section 4.157). The only difference is that, if any array element contains a smaller number of bytes than specified by the mask string, that sub-field will be padded to the designated length with spaces, rather than nulls.

PUTVECSPC may only be used with random or unfmt files. An attempt to write to a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.158 PWIDE

Format: PWIDE

Type: statement

Purpose: to print double width characters on the printer

Example: PWIDE

Execution of the PWIDE statement causes subsequent printed output, directed to the printer, to be double width characters, assuming that this feature is supported by the printer hardware.

see also: PNARROW (section 4.144)

4.160 RAD

Format: RAD(x)

Type: intrinsic function

Example: Y = RAD(X)

The function RAD is provided to convert the numeric expression x, expressed in degrees, to radians.

4.161 RANDOMIZE

Format: RANDOMIZE

Type: statement

Purpose: to reseed the random number generator

Example: RANDOMIZE

The RANDOMIZE statement is used to reseed the random number generator. If it is not reseeded, the RND(0) function will return the same sequence of numbers each time that a program is executed.

RANDOMIZE determines the new seed by measuring the cumulative time spent waiting for console input with EDIT$, INKEY$, INPUT, or INPUT$. For RANDOMIZE to be meaningful, one of those input functions must be executed first.

see also: RND (section 4.177)

4.162 READ

Format: READ var {,var}

Type: statement

Purpose: to read constant values from a DATA statement, and assign them to specified variables

Example: READ A,B%,C$

The READ statement is used to assign constant values from one or more DATA statements to specified variables. One value is read from a DATA statement for each variable in the variable list. If the number of variables in the variable list exceeds the number of data items available, an insufficient data error will be generated. Data items may be either string or numeric. A READ of a numeric data item follows the rules of the VAL function. That is, the following two examples produce the identical results.

```
READ X          (or)          READ X$ : X = VAL(X$)
```

Upon program execution, the DATA pointer is initialized to point to the first item in the first DATA statement in the program. When a READ statement is executed, one value is read from the list for each variable specified, and the pointer is advanced to the next data item. When the data items in a DATA statement are depleted, the pointer is positioned at the next DATA statement in the program, such that all data items constitute a contiguous list. The DATA pointer may be repositioned at any time with the RESTORE statement.

see also: DATA (section 4.035) and RESTORE (section 4.173)

4.163 READONLY

Format: READONLY

Type: option

Example: OPEN 3 "DATAFIL.FIL" READONLY

READONLY is an option which may be designated as part of an OPEN statement. A READONLY file is one which may be shared, on a read/only basis, with other users.

see also: LOCKED (section 4.105) and UNLOCKED (section 4.215)

4.164 RECGET

Format: RECGET(x)

Type: intrnsic function

Example: A = RECGET(1)

The RECGET function returns a numeric value equal to the current value of the GET pointer for file number x. The GET pointer may be set explicitly with the GETSEEK statement (section 4.077).

RECGET may only be used with random or unfmt files. An attempt to reference a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.165 RECLEN

Format: RECLEN x

Type: option

Example: OPEN 1 "TESTFILE" RECLEN 212

RECLEN is an option which may be designated as part of an OPEN or CREATE statement. If not included, the file is assumed to be sequential type. If it is, the file is presumed to be random or unfmt type.

The RECLEN option must be immediately followed by a numeric expression which evaluates to the desired logical record length for this file. In the explicit form, it must be greater than zero, and smaller or equal to 250 or the reclen limit specified in a SIZES statement. When opening a random (but not unfmt) file, an implied form of this option is recognized. In this form, a RECLEN of 0 may be specified, and BASIC/Z will supply the correct value. The RECLEN of a file may be retrieved with the RECSIZE function (section 4.168).

When a random file is created, the RECLEN must be explicitly declared. From then on, the RECLEN may never change. An attempt to open the file with any other explicit RECLEN will generate a reclen error.

see also: file handling (section 6)

4.166 RECORD

Format: RECORD x

Type: option

Example: GET 1 RECORD 10 A$

The RECORD option is used with random or unfmt file read/write statements (GET, GETNUM, PUTVEC, etc.) to explicitly specify the logical record number to be accessed.

see also: file handling (section 6)

4.167 RECPUT

Format: RECPUT(x)

Type: intrinsic function

Example: A = RECPUT(1)

The RECPUT function returns a numeric value equal to the current value of the PUT pointer for file number x. The PUT pointer may be set explicitly with the PUTSEEK statement (section 4.155).

RECPUT may only be used with random or unfmt files. An attempt to reference a sequential file will generate a wrong file type error.

see also: file handling (section 6)

4.168 RECSIZE

Format: RECSIZE(x)

Type: intrinsic function

Example: IF RECSIZE(2) > 128 THEN GOTO 2000

The RECSIZE function returns a numeric value equal to the logical record size of the file associated with file number x. This function may be particularly valuable when a file is opened with implied form of the RECLEN option.

RECSIZE may only be used with random or unfmt files. An attempt to reference a sequential file will generate a wrong file type error.

see also: RECLEN (section 4.165) and file handling (section 6)

4.169 REM

Format: REM remark text

Type: statement

Purpose: to allow explanatory text to be included in a program

Example: REM any explanatory text

REM statements are not executed, but are used to include comments and explanations. They will be output exactly as entered when the program is listed. Remarks may be used as freely as desired, since they generate absolutely no additional code in a compiled program.

An exclamation point (!), or a single quote (') may also be used in place of the keyword REM. All characters in a program line which follow a REM, !, or ' are ignored.

4.170 RENAME

Format: RENAME(x$) = y$

Type: statement

Purpose: to rename a disk file

Example: RENAME("OLDFILE") = "NEWNAME"

The RENAME statement is used to change the name of the file x$, to the new name specified by y$. If y$ includes a drive reference, it will be discarded, as the drive is established by x$. The file to be renamed may not be open at the time of execution of this statement.

4.171 REPEAT$

Format: REPEAT$(x$,x)

Type: intrinsic function

Example: A$ = REPEAT$(X$,X)

The function REPEAT$ returns a string comprised of the string expression x$, repeated x number of times.

4.172 RESET

Format: RESET [x]

Type: statement

Purpose: to reinitialize the operating system after disk changes

Example: RESET 2R101

The RESET statement is used to inform the operating system that one or more disks have been replaced. If RESET is executed with no argument, the entire disk system is reset. If the numeric expression x is included, it is used as a bit pattern to select which drives to reset. The least significant bit corresponds to drive A, while the sixteenth bit corresponds to drive P. For example, an argument of 5 (equivalent to 2R101) would reset only drives A and C.

Before a disk is removed, it is mandatory that any open files on that drive be properly closed with the CLOSE statement (section 4.016). The RESET statement is executed only after the disks are replaced, and the affected drives are ready.

If a disk is replaced without executing RESET, a subsequent attempt to write to that drive will generate a read only disk error.

4.173 RESTORE

Format: RESTORE [dest]

Type: statement

Purpose: to allow DATA statements to be reread from a specified starting point

Example: RESTORE 3000

The RESTORE statement is used to position the data pointer for subsequent READ statements.

If a label or line number is not specified, the pointer will be set to the first item in the first DATA statement in the program. If a line number is specified, the pointer will be set to the first DATA item in that line. If a label is specified, the pointer will be set to the first DATA item following the label location, although not necessarily on the same line.

4.174 RESTORERR

Format: RESTORERR

Type: statement

Purpose: to restore the global error trap destination address saved with SAVERR

Example: RESTORERR

The RESTORERR statement is used to restore the global error trap destination address temporarily saved with SAVERR.

see also: SAVERR (section 4.181) and error handling (section 7)

4.175 RETURN

Format: RETURN

Type: statement

Purpose: to terminate a subroutine

Example: RETURN

The RETURN statement transfers control to the statement at the location saved as the top entry on the subroutine stack. This will be the most recently executed of:

1) the statement following the most recently executed GOSUB
2) the statement at the argument to the most recently executed PUSH

If a RETURN is executed with no locations saved on the subroutine stack, an invalid return error will be generated.

4.176 RIGHT$

Format: RIGHT$(x$,x)

Type: intrinsic function

Example: A$ = RIGHT$(X$,X)

The function RIGHT$ will return a string comprised of the rightmost x characters of the string expression x$. The numeric expression x must evaluate in the range of 0 through 250. If x is greater than LEN(x$), the entire string x$ will be returned. If x equals zero (0), a null string is returned.

see also: LEFT$ (section 4.099) and MID$ (section 4.115)

4.177 RND

Format: RND(x)

Type: intrinsic function

Example: A = RND(X)

RND will return a pseudo random number between the value of 0 and 1. The numeric argument x is mandatory, as it controls the random number generation.

If x=0, the last random number generated is used as the seed. Repeated calls to RND with a zero argument will return a pseudo random sequence of numbers.

If x is non-zero, the random number is generated using x as the seed. Repeated call to RND with a specific non-zero argument will always return the same value.

see also: RANDOMIZE (section 4.161)

4.178 RUN

Format: RUN [x$]

Type: statement

Purpose: to optionally load a BASIC/Z object program from disk, clear certain data items, and execute the program

Example:
```
IF A >= 100 THEN RUN
RUN "B:NEXTPGM"
```

BASIC/Z recognizes two distinct versions of the RUN statement. Generally, RUN is used to load a compiled object program from disk, reset all variables and certain other data, and then execute the program. However, if the optional expression x$ is not included, then no program is loaded from disk, but the same data is reset and the current program is restarted from the first executable statement.

Upon execution of the object program, the following actions are initiated:

1) Dynamic arrays are erased, all numeric variables are reset to zero (0), and all strings are reset to null ("").
2) Any global error trap (ON ERROR GOTO) is reset.
3) Any local error traps (END or ERROR destinations for files) are reset
4) Subroutine, function, and expression evaluation stacks are cleared.
5) Data pointer for READ statements is restored to the first data item in the program.
6) All definitions (DEF FA) of entry points to assembly language subroutines are cleared.

All other data and conditions remain unchanged. Among other items, it should be specifically noted that the direction of the print stream and the value of the current string delimiter are unchanged, as well as the memory end (set by MEMEND) and any assembler subroutines loaded above that address.

Unlike most other languages, files may remain open during execution of a CHAIN or RUN statement, offering a substantial speed advantage. While error traps local to each file are reset, they may be re-established with ON END x GOTO or ON ERROR x GOTO, which are described in sections 4.127 and 4.129.

If the argument x$ is included, it must evaluate to a string containing an optional drive and name of a compiled BASIC/Z object program to be loaded and executed. The file type (.BZO) is assumed, so it need not be included.

If it is desired to pass any variable data between the programs, CHAIN (section 4.012) should be used rather than RUN.

4.179 RVIDEO

Format: RVIDEO

Type: statement

Purpose: to cause console output to be reverse video

Example RVIDEO

Execution of the RVIDEO statement causes subsequent printed output, directed to the console, to be reverse video, assuming this feature is supported by the console hardware.

see also: NVIDEO (section 4.124)

4.180 SAVE

Format: SAVE x$, x, y

Type: statement

Purpose: to save a block of memory as a disk file

Example: SAVE "PROGRAM$", 16R9000, 16R9120

Upon execution, SAVE will cause a block of memory from location x through location y to be saved as a newly created disk file.

The expression x$ must evaluate to a string containing an optional drive name (if other than the current default), and the complete name of the new file to be created.

The numeric expressions x and y must evaluate to valid starting and ending addresses for the memory block to be saved.

Due to characteristics of the operating system, files may only be saved in even 128 byte blocks. Therefore, a statement to save 129 bytes would actually create a file of 2 records, or 256 bytes. This must be carefully considered if the file is to be retrieved with a LOAD statement, as other needed data could be inadvertantly over-written.

4.181 SAVERR

Format: SAVERR

Type: statement

Purpose: to temporarily save the global error trap address

Example: SAVERR

The SAVERR statement is used to temporarily save the global error trap destination address, for later recall with RESTORERR. This allows a program to temporarily change the error trap destination, with easy restoration to the original.

The SAVERR function provides only one level of storage, that of the most recent execution.

see also: RESTORERR (section 4.174) and error handling (section 7)

4.182 SCRATCH

Format: SCRATCH x$

Type: statement

Purpose: to erase one or more disk files

Example: SCRATCH "B:OLDFILE.DAT"

The SCRATCH statement is used to erase one or more disk files which match the expression x$. The expression must evaluate to a string containing an optional drive name (if other than the current default), and the name of the file(s) to be erased. It may contain ambiguous references. A question mark (?) will match any character, and an asterisk (*) will match any name or type.

4.183 SEARCH

Format: SEARCH(avar, avar, relop, expr [,x])

Type: intrinsic function

Example: X# = SEARCH(X&(2), X&(21), >=, Y&)

The numeric function SEARCH is provided to scan through designated elements of an array to find the first element which matches an expression, using any relational operator (=, <>, <, >, <=, >=).

Upon execution, the designated array is scanned from the first avar, through the second avar to find the first element which matches the expr using the specified relop. A numeric value is returned to indicate the relative position of the first match, or zero if no match is found.

In the above example, the array X& is scanned from element 2 through 21, to locate the first element which is greater or equal to Y&. If all of those elements were less than Y&, 0 would be returned. However, if X&(3) were >= Y&, the value 2 would be returned, to indicate that a match was found on the second element compared.

When SEARCH is used with a string array, an optional length limit argument may also be included. This option will limit the comparison to the first x characters of each array element. For example:

```
pointer# = search( x$(0), x$(99), = , "ABC", 3 )
```

In the above example, the array x$ would be searched, to locate an element in which the first three characters equal ABC, regardless of the total length of that element.

4.184 SELECT

Format: SELECT x$

Type: statement

Purpose: to select a default drive

Example: SELECT "B:"

The statement SELECT is used to specify a default drive for future disk operations. The drive to be selected is determined by the first character in the string expression x$, which must evaluate to a letter in the range of A through P.

4.185 SELECT$

Format: SELECT$

Type: intrinsic function

Example: X$ = SELECT$

The function SELECT$ returns a two byte string containing the name of the current default drive, followed by a colon (:).

4.186 SERIAL

Format: SERIAL

Type: intrinsic function

Example: IF SERIAL <> 12345 THEN STOP

Then function SERIAL returns a numeric value equal to the serial number of the RUN/Z Run Time Library then in use.

4.187 SETERROR

Format: SETERROR x

Type: statement

Purpose: to simulate or force an error condition

Example: SETERROR 22

The SETERROR statement is used to simulate the occurrence of a BASIC/Z error, or to force a program defined error condition. The argument x defines the error code number, and must evaluate in the range of 0 through 255.

As detailed in appendix C, BASIC/Z errors may be sub-classified as local errors (those pertaining to a specific disk file) or global errors (comprising all the rest). All the remaining codes, through 255, are available for user definition. However, it is best to use the highest numbers possible, to avoid any conflict with BASIC/Z error codes which may be added in the future.

Upon execution of SETERROR, the program will react just as though an error had occurred. If no error traps have been enabled, the appropriate error message will be displayed, and the program terminated. If the code is user-defined, the message "Error" is displayed on the console.

If the code is that of a local error, and the most recently accessed file has local error trapping (or end file trapping, if an end file error) enabled, then control will be transferred to that designated routine. In all other cases, control will be transferred to the global error trapping routine specified by the most recently executed ON ERROR GOTO statement.

see also: error handling (section 7)

4.188 SETUP

Format: SETUP

Type: intrinsic function

Example: IF PEEK(SETUP+X) <> 0 THEN 1000

The function SETUP returns a numeric value equal to the base address of the user configuration area. An application program can alter or interrogate memory locations at designated offsets to this base address, to take advantage of a wide variety of system features.

A complete definition of the user configuration area will be found in appendix D.

4.189 SGN

Format: SGN(x)

Type: intrinsic function

Example: ON SGN(X)+1 GOTO 100, 200, 300

The SGN function returns -1 if the sign of x is negative, +1 if the sign of x is positive, or 0 if x equals zero.

4.190 SIN

Format: SIN(x)

Type: intrinsic function

Example: A = SIN(X)

The SIN function returns a numeric value equal to the sine of the expression x.

The argument x must be expressed in radians. Degrees may be converted to radians by dividing the number of degrees by 2pi, or with the function RAD, described in section 4.160.

4.191 SIZE

Format: SIZE(x)

Type: intrinsic function

Example: IF SIZE(1) > 99 THEN 500

The SIZE function returns a numeric value equal to the number of records in the file associated with file number x.

If the file is either random or unfmt type, the value refers to the number of logical records of the specified reclen. If sequential, it refers to the number of physical, 128 byte records.

4.192 SIZES

Format: SIZES(nlit, nlit, nlit, nlit)

Type: statement

Purpose: to specify the number of storage bytes for each variable type, and the reclen limit for all files

Example: SIZES(8,4,100,512)

Each argument in a SIZES statement must appear as a numeric literal, delimited by a comma.

The first nlit specifies the number of bytes of storage for each real (floating point) variable, known as RSIZE. It must be greater than ISIZE and less than 11.

The second nlit specifies the number of bytes of storage for each BCD integer, known as ISIZE. It must be greater than 2 and less than RSIZE.

The third nlit specifies the default maximum string length, known as SSIZE. It must be greater than 0 and less than 251.

The fourth nlit specifies the maximum reclen which may be used in any random or unfmt file in this program. It must be greater than 249.

If a SIZES statement appears in any program which is part of a CHAIN, then every program in the CHAIN must contain an identical SIZES statement.

If a SIZES statement is not explicitly included in a program, BASIC/Z will presume default SIZES of (5,3,40,250).

see also: integer variables (section 2.012), real variables (section 2.013), string variables (section 2.014), and RECLEN (section 4.165)

4.193 SORT

Format: SORT avar, avar, x$, x

Type: statement

Purpose: to sort an array

Example: SORT TXT$(0), TXT$(1942), "DU", 14

The SORT statement recognizes all data types in BASIC/Z. It may be used to sort any memory array to the following specifications:

ARG 1 - LOW ELEMENT:

The first argument must name the lowest element of the array to be sorted. It need not be element 0, as any number of elements may be "skipped over".

ARG 2 - HIGH ELEMENT:

The second argument must name the highest element of the array to be sorted. It must name the same array as argument one, and define an element number which is greater than that defined by argument one.

ARG 3 - OPTION STRING:

The third argument must evaluate to a string which defines the sequence and comparison options. The sequence option may be specified by including an A (for ascending) or D (for descending). If neither is present, the default is to ascending sequence. The comparison option is valid for string sorts only. If a U is included, lower case characters will be forced to upper case for comparison only. If an L (or neither) is present, all characters will be treated "as-is".

ARG 4 - LIMITED KEY:

The fourth argument must evaluate to a numeric in the range of 1 through 250. Although it is used in string sorts only, the argument must always be present, even if just as a "place-holder". To disable this option, include a value of 250 in this position. If a limited key length of x is designated, only the first x characters of each string will be compared. For example, if x = 7, the strings "Chicago" and "Chicago Heights" would be considered equal.

4.194 SPACELEFT

Format: SPACELEFT

Type: intrinsic function

Example: A = SPACELEFT

The function SPACELEFT returns an integer numeric value, equal to the number of unused bytes of memory at the time it is executed.

4.195 SPC$

Format: SPC$(x)

Type: intrinsic function

Example: PRINT SPC$(20)

The SPC$ function returns a string consisting of x spaces. The numeric expression x must evaluate in the range of 0 through 250. Although REPEAT$ could also be used for this purpose, the function SPC$ is considerably more efficient.

4.196 SPOOL

Format: SPOOL x

Type: statement

Purpose: to direct printed output to a file

Example: SPOOL 19

The SPOOL statement is used to designate that printed output should be directed to the sequential file associated with file number x. The primary purpose of SPOOL is to allow a printed report to be saved temporarily in a disk file, for later printing on the printer.

While a SPOOL statement is "in effect", all print functions (such as TAB, FORMFEED, etc.) assume attributes of the system printer. At the time SPOOL is executed, PCOL and PLINE counters are reset to 1, as it is assumed that printing will start at "top-of-form". However, the previous values are saved and restored when the SPOOL is completed, so that no printer data is lost.

see also: CONSOLE (section 4.024), ECHO (section 4.046), LPRINTER (section 4.110), and NULL (section 4.196)

4.197 SQR

Format: SQR(x)

Type: intrinsic function

Example: A = SQR(X)

The SQR function returns a numeric value equal to the positive square root of the numeric expression x. The argument x must evaluate to a positive number.

4.198 STATUS

Format: STATUS(x)

Type: intrinsic function

Example: IF STATUS(2) THEN 1000

The STATUS function is used to test the open/closed status of file number x. It returns true (1) if the file is open, or false (0) if the file is closed.

4.199 STDINT

Format: STDINT

Type: statement

Purpose: to cause console output to be standard intensity

Example: STDINT

Execution of the STDINT statement causes subsequent printed output, directed to the console, to be standard intensity. STDINT mode is the default condition, and need not be explicitly initialized.

see also: HIGHINT (section 4.084) and LOWINT (section 4.109)

4.200 STEP

Format: STEP x

Type: option

Example: FOR Z = 0 TO 100 STEP 2

STEP is utilized to designate the numeric value to be algebraically added to the counter variable upon execution of a NEXT statement. If the STEP option is not included, BASIC/Z will supply a default STEP value of plus one (+1).

see also: FOR (section 4.070) and NEXT (section 4.120)

4.201 STOP

Format: STOP

Type: statement

Purpose: to terminate execution of a program

Example: STOP

Upon execution of a STOP statement, the program will be terminated, and control will be returned to the disk operating system. In BASIC/Z, this is functionally identical to the END statement, which is described in section 4.049.

4.202 STR$

Format: STR$(x)

Type: intrinsic function

Example: A$ = STR$(X)

The STR$ function returns a string representation of the value of the numeric expression x.

In constructing this string, BASIC/Z follows the rules detailed under Numeric output formats (section 2.024). Of particular note is that the first position is set to a space or a minus sign, and a trailing space is always appended. For explicit numeric positioning, BASIC/Z provides the FMT function (section 4.066).

4.203 STRING

Format: STRING x$

Type: statement

Purpose: to define the current string delimiter

Example:
```
STRING CHAR$(16RFF)
STRING ""
```

The STRING statement defines the current string delimiter used to separate a sub-field when accessed by a GET, GETFIELD, or INPUT statement. The default string delimiter is the comma (,).

If x$ evaluates to a null string, then there is no string delimiter. This feature allows strings with all characters (from 0 to 255) to be read with a GET statement. However, in this case a GET statement can have only one string variable in the variable list, as a record can contain only one undelimited string.

4.204 STRING$

Format: STRING$(cavar,cavar)

Type: intrinsic function

Example: ANY$ = STRING$(X&(4), X&(22))

The STRING$ function returns a string comprised of characters having the ascii code(s) extracted from the designated control array, starting at the first cavar, and ending at the second cavar.

```
100     x&(1) = 65
110     x&(2) = 66
120     x&(3) = 67
130     x$ = string$(x&(1),x&(3))
```

After execution:

```
x$ = "ABC"
```

When used in conjunction with FILL, FILLSPC, GETVEC, and PUTVEC, STRING$ provides a powerful yet easy means of blocking and de-blocking data records.

see also: FILL (section 4.063), FILLSPC (section 4.064), GETVEC (section 4.079), PUTVEC (section 4.157), and Array variables (section 2.016)

4.205 SWAP

Format: SWAP var, var

Type: statement

Purpose: to exchange the values of two variables

Example: SWAP X%, Y#

The SWAP statement is used to exchange the values of any two variables which match in general type (i.e. numeric or string). However, the numeric variable types need not be identical, as long as the values assigned are in range for the variables.

4.206 TAB

Format: TAB(x [,y]);

Type: print function

Example:
```
PRINT TAB(10); A$
PRINT TAB(10,10); A$
```

The TAB function is used to explicitly set a print position. As a print function, it is legal only in a PRINT statement, and in all cases must be followed by a semi-colon (;) delimiter.

BASIC/Z recognizes two versions of the TAB function. Although they are related, they are syntactically distinct, and distinguished by the number of arguments included. In all cases, TAB argument values are indexed to one (1).

If the TAB function has just one argument, the print position will be moved to the column specified by the argument, by filling with spaces. If the print position is already beyond that specified, the function will be ignored.

If the TAB function has two arguments, absolute cursor positioning will be utilized. In this case, the first argument specifies the line position, while the second argument specifies the column position. It should be noted that the home position would be addressed as TAB(1,1). If the print stream is directed to the printer, and the print position is already beyond that specified, the function will be ignored.

4.207 TAN

Format: TAN(x)

Type: intrinsic function

Example: A = TAN(X)

The TAN function returns a numeric value equal to the tangent of the expression x.

The argument x must be expressed in radians. Degrees may be converted to radians by dividing the number of degrees by 2pi, or with the function RAD, described in section 4.160.

4.208 TERMPOS

Format: TERMPOS

Type: intrinsic function

Example: TEXT$ = EDIT$(TEXT$,MAX.LEN&,TERMPOS)

The function TERMPOS is used to determine the relative position of the cursor, within the response line, at the time of the most recent termination of an INPUT$ or EDIT$ function. In case of an interrupt, to display help messages or other tasks, your program may use this information to execute an EDIT$ function at the exact position it was exited. The value returned by TERMPOS will be numeric, in the range of 0 through 250.

see also: EDIT$ (section 4.047)

4.209 THEN

Format: IF x THEN stmt {:stmt} [ELSE stmt {:stmt}]

Type: sub-statement

Purpose: to control program flow in the event that the expression in an IF statement evaluates true

Example: IF A=B THEN C=1

In BASIC/Z, THEN is a mandatory part of every IF statement. For a complete description, please refer to IF (section 4.085).

4.210 TO

Format: FOR nvar = x TO y [STEP z]

Type: sub-statement

Example: FOR A = B TO C

In BASIC/Z, TO is a mandatory part of every FOR statement. For a complete description, please refer to FOR (section 4.070).

4.211 UCASE$

Format: UCASE$(x$)

Type: intrinsic function

Example: TEXT$ = UCASE$(TEXT$)

The function UCASE$ returns the argument expression x$, but with every lower case letter converted to the upper case equivalent. All other characters remain unchanged.

see also: LCASE$ (section 4.098)

4.212 ULINE

Format: ULINE

Type: statement

Purpose: to cause console output to be underlined

Example: ULINE

Execution of the ULINE statement causes subsequent printed output, directed to the console, to be underlined, assuming this feature is supported by the console hardware.

see also: NONLINE (section 4.122)

4.213 UNFMT

Format: UNFMT

Type: option

Example: OPEN 1 "FILE.DAT" RECLEN 128 UNFMT

UNFMT is an option which may be used as part of a CREATE or OPEN statement. It specifies a simplified form of random file which eliminates the header record. This allows compatibility with other languages, but may introduce certain errors in the PUT and EOF pointers when the logical record length is set to less than 128. Also, in the UNFMT mode, the logical record length and logical file size (if altered with the EOF statement) are not maintained by the system, but just discarded when the file is closed.

see also: CREATE (section 4.026), OPEN (section 4.134), and file handling (section 6)

4.214 UNLOCK

Format: UNLOCK(x,y)

Type: statement

Purpose: to unlock records previously locked

Example: UNLOCK(1,42)

The UNLOCK statement is used to unlock record number y in file number x. Generally, it is presumed that the record was locked previously with the LOCK function, but this is not mandatory. Any UNLOCK request with valid arguments is considered to be successful.

see also: LOCK (section 4.104)

4.215 UNLOCKED

Format: UNLOCKED

Type: option

Example: OPEN 1 "FILE" UNLOCKED

UNLOCKED is an option which may be designated as part of an OPEN or CREATE statement. An UNLOCKED file is one which may be shared, on a read/write basis, with other users.

see also: LOCKED (section 4.105) and READONLY (section 4.163)

4.216 UNTIL

Format: UNTIL expr

Type: statement

Purpose: to terminate a DO/UNTIL loop

Example: UNTIL INDEX& > 20

The UNTIL statement is used to terminate a DO/UNTIL loop. A complete definition of this structure may be found under DO (section 4.045).

4.217 UPCASE

Format: UPCASE

Type: statement

Purpose: to force alphabetic console input to upper case

Example: UPCASE

BASIC/Z provides a software "cap-lock" option for executing programs. If this feature is enabled, through execution of an UPCASE statement, all alphabetic console input is forced to upper case, as it is typed. Execution of the statement LOWCASE allows, but does not force, lower case alphabetic input.

see also: LOWCASE (section 4.108)

4.218 USER

Format: USER x

Type: statement

Purpose: to alter the current user number

Example: USER 7

The USER statement is used to alter the current user number to that specified by the numeric expression x. Depending upon the operating system in use, a valid user number may range from 0-15 or from 0-31.

The current user number may be accessed from the user configuration area of memory, which is described in appendix D of this manual.

4.219 VAL

Format: VAL(x$)

Type: intrinsic function

Example: A = VAL(X$)

The function VAL converts the string expression x$ to a number, and returns the numeric value of it. It ignores leading spaces, and will recognize radix format and scientific notation, in addition to normal decimal numbers. However, VAL provides no expression evaluation.

Examples:

```
val("A%+B%")   = 0
val("2+2")     = 2
val("16R1F")   = 31
val("  21QL") = 21
```

4.220 VARPTR

Format: VARPTR(var)

Type: intrinsic function

Example: X# = VARPTR(A$(21))

The function VARPTR returns a numeric value equal to the absolute memory location of the designated variable.

4.221 VERIFY

Format: VERIFY(x$,y$)

Type: intrinsic function

Example: A = VERIFY(X$,Y$)

VERIFY ascertains if all characters comprising x$ are also found in y$, without regard to their relative position. A numeric value is returned to show the position of the first character in x$ which is not also found in y$. If all characters in x$ are also present in y$, it returns 0.

4.222 VIDEO

Format: VIDEO(x)

Type: statement

Purpose: to ouput a user-defined series of control codes to the console

Example: VIDEO(7)

The VIDEO statement may be used to control any special video attributes which may be available, such as character size, key click, etc. under program control. Upon execution, a user-defined series of control codes is output to the console. This is generally preferable to use of a PRINT statement, to avoid hardware dependence, and to avoid the possible introduction of errors in the CCOL and CLINE functions.

Up to eight (8) sets of control codes may be defined with the INSTALL utility program. Depending upon the value of the expression x (which must range between 0 and 7), the corresponding set of control codes will be output.

see also: PRINTER (section 4.149)

4.223 WEND

Format: WEND

Type: statement

Purpose: to terminate a WHILE/WEND loop

Example: WEND

The WEND statement is used to terminate a WHILE/WEND loop. A complete definition of this structure may be found under WHILE (section 4.226).

4.224 WHEN

Format: WHEN expr

Type: statement

Purpose: to conditionally execute a block of program lines

Example:
```
WHEN FLAG& <> 0
 PRINT "FLAG IS NOT ZERO"
 NEXT.FLAG& = 0
WHEND
```

A WHEN/WHEND construct may be used to provide conditional execution of a block of program lines, based upon the result of a logical expression. If the expression following WHEN evaluates true, then statements within the block are executed. If the expression evaluates false, then control is transferred to the first statement after the associated WHEND statement.

WHEN and WHEND must always be used together, and may be nested up to 255 levels.

4.225 WHEND

Format: WHEND

Type: statement

Purpose: to terminate a WHEN/WHEND structure

Example: WHEND

The WHEND statement is used to terminate a WHEN/WHEND structure. A complete definition may be found under WHEN (section 4.224).

4.226 WHILE

Format: WHILE expr

Type: statement

Purpose: to repeatedly execute a series of statements, as long as the expression following WHILE is true

Example:
```
WHILE INDEX% <= 20
 READ ITEM(INDEX%)
 INCR INDEX%
WEND
```

BASIC/Z provides three looping structures (DO/UNTIL, FOR/NEXT, and WHILE/WEND), to allow conditional repetition of a set of statements. WHILE and WEND statements are always used together, and provide a means to execute the set of statements zero or more times. While this structure is quite similar to a DO/UNTIL loop, note that with DO/UNTIL the statements are always executed at least once.

If the expression following WHILE evaluates true, statements within the loop are executed until the associated WEND is reached. That causes an unconditional branch back to the WHILE, and the expression is again evaluated. This loop continues until the expression evaluates false, causing a branch to the first executable statement following the associated WEND. The statements within the loop must modify the value of the expression, or the loop will be executed endlessly.

WHILE/WEND loops may be nested up to 255 levels.

see also: DO (section 4.045) and FOR (section 4.070)

5.001 BASIC functions

A BASIC/Z program may contain references to intrinsic functions (those which are pre-defined as reserved words) as well as functions which are user-defined (those which are defined by the programmer). BASIC/Z provides an extensive facility for user-written functions, which will allow you to add your own special functions to the language. In BASIC/Z, user-defined functions may be single line, or the more powerful multi-line variety. Unique to BASIC/Z is the fact that a multi-line function is recursive (i.e. it may reference itself while retaining the values of local variables at each level). Several examples will be found later in this section.

Each user-written function which is referenced must also be defined within the program, using a DEF FN statement. However, unlike some other languages, the physical location of the definition is not significant. The name of a function is FN followed by a variable name (a letter, optionally followed by any combination of letters, numbers, and periods). Every function must return data to the expression which referenced it. The function name must agree in type with the data to be returned (i.e. if the function returns a string, the name must end with a dollar sign ($) as a type identifier, while functions which return numeric data have no trailing type identifier). A function name may be defined only once within a program.

User-defined functions may be thought of as a type of subroutine, although they offer some special advantages. You may optionally pass the value of up to four expressions (either numeric or string) to the function at the time it is executed. These values are assigned to special local variables, and are referenced by the names of the dummy arguments in the function definition. The data type of the function name does not limit what types of data may be passed to the local variables in the function. Any constants, expressions, or other functions may be used, as long as they match the number and type of arguments in the definition. Dummy arguments may take any scalar (non-array) variable name.

In a function definition, both local variables and global variables may be referenced. Any variable named by a dummy argument is considered local to the function. As such, it is unrelated to a variable by the same name outside of the function definition. All other variables are considered to be global.

SINGLE-LINE FUNCTIONS:

A single-line function takes the following form:

```
DEF FNname [(dummy {,dummy})] = expr
```

When a single-line function is referenced, every occurrence of a dummy argument (if any) in the expression is replaced by an appropriate value passed by the reference. Next, the expression is evaluated, and the result is returned to the calling expression.

Some examples of single-line functions:

```
100 DEF FNtotal.pay = fnround(hourly.rate * hours.worked)
110 DEF FNround(x) = int(x*100+0.5)/100
120 DEF FNmodulo(x,y) = x-(y*int(x/y))
130 DEF FNinteger.part(x) = sgn(x)*int(abs(x))
```

The example in line 100 above illustrates a useful function, even though no values are passed to it by the function reference. The value returned is always the correct total pay, which automatically reflects any changes in either of the values used to compute it. While the same information might be maintained in another variable named total.pay, extra programming overhead would be needed to see that the value was always current.

You may note that the examples in lines 120 and 130 above actually duplicate two intrinsic functions in BASIC/Z, to help illustrate their operation. Any reference to fnmodulo(x,y) would produce the identical result as MOD(x,y), while fninteger.part(x,y) duplicates FIX(x,y).

MULTI-LINE FUNCTIONS:

Multi-line functions take the following form:

```
DEF FNname [(dummy {,dummy})]
stmt
{stmt}
FNEND[$] = expression
```

When a multi-line function is referenced, the local variables named by dummy arguments (if any) are assigned the appropriate values passed by the function reference. Next, the statements within the body of the function are executed. These statements may reference and assign new values to both local and global variables. References to all functions are allowed. In fact, since multi-line functions are recursive, it may even reference itself while retaining distinct local variables at each level.

The FNEND statement is used to terminate a numeric multi-line function, while FNEND$ terminates a string function. When it is reached, the expression to the right of the equal sign (=) is evaluated, and the result is returned to the calling expression. The terminating expression may contain unlimited references to both local and global variables. Generally, a multi-line function should exit only through the FNEND[$] statement.

The following statements are illegal in a user-defined function:

CHAIN	DEF FN	LINK
COMMON	DDIM	POP
DATA	DIM	PUSH
DEBUG	ERASE	RUN

In addition, the following statements should generally be avoided in a user-defined function:

FOR	GOTO	ON x GOSUB
GOSUB	NEXT	ON x GOTO

Some examples of multi-line functions:

```
rem  fnfactorial returns the factorial of a numeric value

def fnfactorial(x)
when x <= 1
     y = 1
whend
when x > 1
     y = fnfactorial(x-1) * x
whend
fnend = y
```

```
rem  fnltrim$ returns a string with leading spaces removed

def fnltrim$(x$)
while left$(x$,1) = " "
     x$ = right$(x$,len(x$)-1)
wend
fnend$ = x$
```

```
rem  fnrtrim$ returns a string with trailing spaces removed

def fnrtrim$(x$)
while right$(x$,1) = " "
     x$ = left$(x$,len(x$)-1)
wend
fnend$ = x$
```

```
rem  fntrim$ returns a string with both leading and trailing
rem  spaces removed

def fntrim$(x$)
x$ = fnltrim$(x$)
x$ = fnrtrim$(x$)
fnend$ = x$
```

5.002 Assembly language functions

BASIC/Z allows the definition of assembly language functions, to provide linkage to assembly language subroutines. The usual purpose is to perform functions that would be difficult or even impossible in BASIC. This unique feature allows you to design your own extensions to BASIC/Z, which are implemented at the most efficient, assembly language level.

Each assembly language function which is referenced must also be defined within the program, using a DEF FA statement. The name of an assembly language function is FA followed by a variable name (a letter, optionally followed by any combination of letters, numbers, and periods). The function name must agree in type with the data to be returned (i.e. if the function returns a string, the name must end with a dollar sign ($) as a type identifier, while functions which return numeric data have no trailing type identifier). Up to 32 assembly language functions may be defined in a BASIC/Z program.

In addition to naming an assembly language function, the DEF FA statement specifies the absolute execution address of the subroutine. The argument specifying the address is evaluated dynamically, so the statement must be executed prior to any reference. Execution addresses may be re-defined as many times as desired.

Generally, executable code for an assembly language function is brought into memory from disk with the LOAD statement. It should normally be located in a block of memory reserved with a MEMEND statement, above that used by BASIC/Z, but below that occupied by the operating system. The ENDMEM function may be used to determine the highest available memory location. A somewhat unorthodox alternative is to store machine code instructions within a control array, which may then be executed! An example of this concept is demonstrated later in this section.

Since assembly language subroutines are implemented as named functions, they must always return a value to the calling expression, even if it is the system supplied default of zero (0) for numeric functions, or null ("") for string functions. A reference to a function consists of the name, optionally followed by up to four argument expressions, enclosed in parenthesis. For example:

```
PRINT FAAPPLE
A$ = FATEST$(A$, B*C%+3, C, "hello")
X# = FASPECIAL(PTR#(0))
```

If argument expressions are included, they are evaluated individually, and passed to the function by value, via pointers to special temporary buffers. That is, in the third example above, the function would not be passed the memory address of the variable PTR#(0), but rather the address of a temporary buffer which contains the current value of that variable. To pass the actual variable location, the following format would be used instead:

```
X# = FASPECIAL(VARPTR(PTR#(0)))
```

When the assembly language subroutine gains control, the HL register pair contains the address of the linkage table (described below), and the DE register pair contains the address of a 253 byte result buffer. If a result is not explicitly inserted into the result buffer by the subroutine, BASIC/Z will supply a value of zero (0) for numeric functions, or null ("") for string functions. Up to eight levels (16 bytes) of stack space may be used. If more is required, it must be handled by the subroutine. When the subroutine is finished, control is returned to your BASIC/Z program by executing a machine code RET instruction (C9 hex).

LINKAGE TABLE:

Name	Bytes	Description
@ARG^1	2	Pointer to argument one
@ARG^2	2	Pointer to argument two
@ARG^3	2	Pointer to argument three
@ARG^4	2	Pointer to argument four
@ARG^CNT	1	Number of arguments passed (0-4)
@CSIZE	1	Number of storage bytes for each control variable (always 1)
@CDSIZE	1	Number of storage bytes for each control/d variable (always 2)
@RSIZE	1	Number of storage bytes for each real variable (4-10)
@ISIZE	1	Number of storage bytes for each BCD integer variable (3-9)
@SSIZE	1	Maximum length of each string not dimensioned (1-250)

Values in the above table are only to be read by the assembly language subroutine. They may never be modified!

The format of the arguments passed and the result returned is:

Byte 0 - Type indicator

0 - control
1 - control/d
2 - real (floating point)
3 - BCD integer
4 - string

Byte 1-n - Refer to section 2.010 - 2.014 for the internal storage formats.

Following is an example of the assembly language subroutine named as SYSTEM. This may be implemented as an assembler function call to allow direct interaction with the operating system BDOS. Upon execution, it expects to receive one argument of a three byte string, which contains values to be loaded into the CPU registers C, E, and D respectively. After completion of the call to the BDOS, it returns a three byte string, which contains the values returned in CPU registers A, L, and H respectively.

```
D5          SYSTEM  PUSH D          ; save the result address
7E                  MOV A,M         ; retrieve pointer to @arg^1
23                  INX H
66                  MOV H,M
6F                  MOV L,A
23                  INX H           ; point to max length
23                  INX H           ; point to current length
23                  INX H           ; point to the register data
4E                  MOV C,M         ; load up the registers
23                  INX H
5E                  MOV E,M
23                  INX H
56                  MOV D,M
CD 05 00            CALL 5          ; the BDOS entry point
D1                  POP D           ; retrieve the result address
EB                  XCHG
36 04               MVI M,4         ; make it a string result
23                  INX H
36 03               MVI M,3         ; max length of 3
23                  INX H
36 03               MVI M,3         ; current length of 3
23                  INX H
77                  MOV M,A         ; save the registers
23                  INX H
73                  MOV M,E
23                  INX H
72                  MOV M,D
C9                  RET
```

The above example is not necessarily complete, as it should probably include error checking on the argument string for validity, etc. However, it should serve as a good example of the flexibility of this interface to assembly language subroutines.

As mentioned earlier in this section, a somewhat unorthodox method of storage of machine code instructions is demonstrated here. Note that the instructions from the previous example subroutine are stored in DATA statements, to be READ into the control array named system&(n). As elements of a control array are stored sequentially in memory, these instructions may be executed right in the array by defining the base element as the execution address of the function!

While this method must be used with some care, it provides a rather simple and unique method of handling smaller assembler subroutines.

```
data  D5,7E,23,66,6F,23,23,23,4E,23,5E,23,56,CD,05,00,D1
data  EB,36,04,23,36,03,23,36,03,23,77,23,73,23,72,C9
'
dim system&(32)
def fasystem$ = varptr(system&(0))
'
for index& = 0 to 32
read instruction$
system&(index&) = val("16R"+instruction$)
next index&
'
arg$ = chr$(4)          : ' register C = 4 / BDOS punch output
arg$ = arg$ + "A"       : ' register E = 41H / output letter "A"
arg$ = arg$ + chr$(0)   : ' register D = 0
'
result$ = fasystem$(arg$) : ' send "A" to the punch device
```

6,000 File handling

BASIC/Z supports three types of disk files: sequential, random, and UNFMT random. Each file type has certain unique attributes which make it more or less suitable for a particular application. In the following sections, each of these types will be discussed in some detail.

6,001 Sequential files

In a sequential file, variable amounts of disk space are assigned to each logical record. It is generally considered most suitable for saving text as a series of ascii characters. Each logical record occupies exactly the number of bytes defined by the data itself, making it the most efficient file type in terms of disk utilization. However, it is limited in both flexibility, and the speed with which data may be accessed.

The statements and functions used with sequential files are:

ATTR	ERROR	NAME	PUTSEQ
ATTRS	FREE	ON END X	READONLY
CLEAR	GETFIELD	ON ERROR X	SIZE
CLOSE	GETSEQ	OPEN	STATUS
CREATE	INKEY$	PUT$	STRING
END	LOCKED	PUTFIELD	UNLOCKED

Data is written to a sequential file as a series of characters, stored one item immediately after another (sequentially), in the order it is sent to the file. It is read back in the same manner. Generally speaking, each data item must be separated from the next by a special delimiter character, to indicate end of field, end of record, or end of the file itself. BASIC/Z allows a good deal of flexibility here, as the field delimiter is not restricted to a comma (,), as in some other languages. With the STRING statement, it is definable under program control to any other character, or may even be disabled entirely. In fact, two special reserved words (INKEY$ and PUT$) do not recognize any delimiter characters (not even end of file), to allow you complete control of a special sequential file organization. For more information on INKEY$ and PUT$, you may refer to sections 4.090 and 4.152, as the remainder of this section assumes the traditional approach to sequential files.

In a sequential file, a logical record consists of from 0 to 250 bytes of data, terminated by a record delimiter. This record delimiter is assumed to be a carriage return/line feed combination, although the line feed character is optional. The end of file is signified by a SUB character (control-z/1A hex).

The high order bit is not considered significant in recognizing record delimiters. That is, both 0D hex and 8D hex are considered to be the CR record delimiter, while both 0A hex and 8A hex are recognized as the LF character, are are ignored. In all other codes, the high order bit is maintained as a significant part of the data.

A logical record may be further sub-divided into fields, which are marked by the current string delimiter. Sequential data may be read or written a field at a time with the GETFIELD or PUTFIELD statements. The current string delimiter may be altered, or disabled, with the STRING statement.

A new sequential file is generated on disk with the CREATE statement, while OPEN is used to prepare an existing file for access. In both of these cases, the lack of a RECLEN option implies a sequential file type. Sequential files may not be accessed with statements designed for random files. For example, execution of a GETVEC statement with a sequential file will generate a wrong file type error.

A sequential file must be accessed from the first logical record. In order to access record 7, it is necessary to first read all of the preceding records (1 through 6), as this is the only way to determine where record 7 begins. BASIC/Z maintains a single pointer into each sequential file for the entire time it is open. When the file is first created or opened, it points to the first logical record, and advances with each sequential read or write to the file. This pointer may never be "backed-up". The only way to re-access earlier records is to close and re-open the file.

Data may be read from a sequential file in several ways. The INKEY$ function returns a specific number of bytes, with absolutely no regard for field or record delimiters, nor even the logical end of file marker. GETSEQ reads an entire logical record (up to the cr/lf record delimiter), while GETFIELD reads up to the first occurrence of either the current string delimiter or the record delimiter.

Data may be written to a sequential file with PUT$, which writes only the specified characters, appending no delimiters. This allows data fields or records to be constructed "on the fly", so to speak, and also allows easy construction of unusual file structures which do not recognize typical delimiter characters. The PUTFIELD statement is similar, but it automatically appends the current string delimiter, while PUTSEQ appends the cr/lf record delimiter.

When a sequential file is opened, it may be accessed one or more times to read data, optionally followed by one or more write operations. However, the reverse is not true. Once a sequential file has been written to, the pointer is considered to be at the logical end of file, and it may only be further written to, or closed. Likewise, when a sequential file is first created (or opened with a CLEAR option), it is considered to be logically "empty", and therefore can only be written to, not read.

6.002 Random files

Random files will probably be the most often used file type. In a random file, a fixed amount of disk space is assigned to each logical record. A critical advantage of random files is the ability to directly access any record, regardless of its relative location within the file.

The statements and functions used with random files are:

- ATTR	GETNUM	PUT	RECSIZE
- ATTRS	GETSEEK	PUTNUM	SIZE
CLEAR	GETVEC	PUTSEEK	STATUS
CLOSE	GETVEC$	PUTVEC	STRING
CREATE	LOCK	PUTVECSPC	UNFMT
END	LOCKED	READONLY	UNLOCK
- EOF	NAME	RECGET	UNLOCKED
- ERROR	ON END X	RECLEN	
FREE	ON ERROR X	RECORD	
GET	OPEN	RECPUT	

In each OPEN or CREATE statement, the RECLEN option designates the absolute amount of disk space assigned to each record. The entire number of designated bytes are available for data, as BASIC/Z does not insert any delimiters between logical records. When a random file is first created, the record length must be explicitly declared. From then on, it may never change. Any attempt to open the file with another record length will result in generation of a reclen error. The declared record length may be any integer value greater than zero, for which buffer space is available in memory. If a record length of greater than 250 is desired, the maximum record length for all files in the program must be declared in a SIZES statement.

Random data files in BASIC/Z are organized in a rather unusual manner, although it is generally transparent to the programmer. In a random file, the first physical record of 128 bytes is reserved for system information needed by BASIC/Z, but not maintained by the operating system (logical record length and number of records). No special programming action is needed, as BASIC/Z will keep this data updated, and also consider it when addressing a particular logical record. Following this system record, logical data records are "packed" one after another, to maximize disk utilization. For example, with a RECLEN of 16, a file of 32 logical records would occupy precisely 5 physical records on disk (1 system record and 5 data records).

If it is necessary to access data files generated by languages or facilities other than BASIC/Z, a sequential or UNFMT random file type should be used instead.

In BASIC/Z, logical record numbers range from 1 to 65535. Also, the CP/M operating system limits files to a maximum physical or logical size of 8 megabytes. Any attempt to access a record outside of either range will result in a parameter error (error code #6).

A new random file is generated on disk with the CREATE statement, while OPEN is used to prepare an existing file for access. In both of these case, the presence of a RECLEN option combined with the absence of an UNFMT option implies a standard random file. Random files may not be accessed with statements designed for sequential files. For example, execution of a GETFIELD statement with a random file will generate a wrong file type error.

Data from random files may be read with GET, GETNUM, GETVEC or GETVEC$ statements. Likewise, data may be written to a random file with PUT, PUTNUM, PUTVEC or PUTVECSPC statements. Every execution of one of these statements will cause one logical record to be read or written. Since they all access the file in the same manner, the only difference is the nature of the data within the record. Precise definitions of the data formats will be found under each of the above reserved words in section 4 of this reference manual.

Each read or write statement to a random file may include a RECORD option, to directly specify the absolute logical record affected. The records may also be accessed in sequential order by using the indexed form of these statements. If a RECORD option is not included, the indexed form is assumed.

BASIC/Z maintains three pointers into each random file for the entire time it is open. These will be termed the EOF, GET, and PUT pointers.

The EOF pointer contains a value equal to the greatest logical record number ever written to that file (not just the current "session"), unless the file has been logically shortened with an EOF statement, or was opened with a CLEAR option. Any attempt to read a record past the EOF position will generate an end of file error. The current value of the EOF pointer may be accessed with the SIZE function. It may be explicitly changed with the EOF statement, but that will have no effect on the GET or PUT pointers.

The indexed GET pointer is maintained to point to the next record to be accessed by an indexed form of one of the read statements. When a file is opened, the GET pointer is initialized to the first record. After each indexed GET, GETNUM, GETVEC or GETVEC$ statement is executed, the GET pointer is automatically incremented by one. The current value of the GET pointer may be accessed with the RECGET function. It may be explicitly changed with GETSEEK.

The indexed PUT pointer is maintained to point to the next record to be written by an indexed form of one of the write statements. When a file is opened, the PUT pointer is initialized to the last logical record +1, unless a CLEAR option was invoked. Generally, a subsequent indexed PUT will append logical records to the file. After each indexed PUT, PUTNUM, PUTVEC or PUTVECSPC statement is executed, the PUT pointer is automatically incremented by one. The current value of the indexed PUT pointer may be accessed with the RECPUT function. It may be explicitly changed with PUTSEEK.

6.003 UNFMT random files

The UNFMT type of file is very similar in construction and access to a random file. Its primary purpose is to provide portability of data between BASIC/Z and other languages and facilities.

All of the descriptions of statements, functions, pointers, etc. discussed under random files apply equally to UNFMT type files, with the following exceptions:

1- The first physical record of a file is not reserved for system information. Logical record #1 is positioned as physical record #1 in the file.

2- The logical end of file may not be permanently altered with the EOF statement, as the operating system provides no means of recording this information on disk.

3- As the logical record size cannot be maintained by the system, there is no way to test for an invalid reclen when a file is opened. This allows a file to be accessed with different record lengths at different times.

4- Be prepared for the introduction of certain file size errors when the reclen is set to less than 128. Take, for example, the case where one record, of reclen 10, is written to a new file and then closed. Upon the next open, BASIC/Z would see one physical 128 byte record, and be forced to assume it contained twelve 10 byte logical records. The EOF pointer would be incorrectly set to 12, instead of 1, and the PUT pointer would be incorrectly set to 13, instead of 2. Generally, the use of UNFMT files require that an additional control file be used to maintain this type of needed information.

6.004 Multi-user considerations

The major problem facing programmers in a multi-user environment is that of interleaved updating. This situation may occur in an unlocked file mode, when two users update the same file simultaneously, and each receives incomplete information of the other user's update.

For example, assume Dick and Jane are both updating a grocery inventory, in the following sequence:

1- Dick's program reads the apple inventory and finds that 10 apples are currently in inventory.

2- Jane's program reads the apple inventory and finds that 10 apples are currently in inventory.

3- Dick records a purchase of 25 apples, writing the new total of 35 apples to the inventory file.

4- Jane records a sale of 5 apples, writing an apparent new total of 5 apples to the inventory file, rather than the correct total of 30.

The solution to this problem lies in the use of individual record locking, with the LOCK and UNLOCK functions, described in section 4. The key to success is to follow this procedure:

1- Every user must lock a data record before it is read, if there is the slightest chance it may be updated. If another user is accessing it, the lock function will return an unsuccessful result. In this instance, your program should go on to another task, and retry this function at a later time. You should also give consideration to releasing any other records you have locked, as another user may be waiting to access them.

2- When the lock is successful, read the data, and optionally update it. When all operations are complete, be certain to release any locked records with the unlock statement.

Another area of potential conflict arises when two users attempt to extend a file simultaneously. Depending upon the operating system in use, record locking may prove impossible, or ineffective. The simplest solution to this problem lies in pre-allocation. Prior to any use, all shared data files should be filled, to their maximum attainable size, with a value which your program will recognize as a null, or empty record. You may also wish to maintain a control file, with a pointer to the next available data record.

7.000 Error handling

BASIC/Z has been designed to offer extraordinary error handling capabilities. Frankly, the most sophisticated application program imaginable is reduced to little more than rubble when an unexpected error brings all to a screeching halt! With BASIC/Z, the tools you need are available - review them carefully, but most important of all - - use them!

The statements and functions used in error handling are:

CLRAUTO	ERR	LASTFILE	RESTORERR
CLRFN	ERR$	ON END X	SAVERR
CLRNONE	ERRADDR	ON ERROR X	SETERROR
CLRSUB	ERRLINE	ON ERROR CLEAR	
END	ERROR	ON ERROR GOTO	

A complete description of each of these reserved words will be found in section 4 of this manual. Please pay particular attention to the four statements which handle manipulation of the run-time stacks, as proper handling in an error condition is crucial. These are: CLRAUTO (4.017), CLRFN (4.018), CLRNONE (4.019), and CLRSUB (4.020). You may also wish to refer to appendix c for a complete listing of all BASIC/Z error codes.\

7.001 System error handling

When a run-time error occurs, BASIC/Z requires that some special action be taken to ensure that the program logic handles it appropriately. Generally this is done by setting a local or global error trap. However, if an error is generated with no error trap in effect, the following occurs:

1- Execution of the program is interrupted, and a descriptive message (from the table in appendix c) is displayed on the console, followed by a numeric value equal to the object code address (in hex notation) of the statement which caused the error. For example:

```
Disk i/o error at 81FC
```

2- If line number have been compiled into the program with a DEBUG "L" option, then the line number will also be included, such as:

```
Disk i/o error in line 11230 at 81FC
```

3- If you attempt to execute a compiled program, from the operating system level, and it cannot be located, the following message is displayed, with a special error address of 0000:

```
File not found at 0000
```

4- Finally, program execution is terminated, and control is returned to the operating system.

7.002 Error trapping priority

BASIC/Z supports two distinct levels of error trapping, which will generally be termed local and global. Local errors are defined as those with an error code number (see appendix c) of less than 16, while global errors encompass all run-time errors.

Local errors all relate to file handling functions, and are always specific to a particular file. Local error trapping is enabled by including an END or ERROR option in the OPEN or CREATE statement, or by executing an ON END x GOTO or ON ERROR x GOTO statement. Local error trapping, if enabled, always takes precedence over global error trapping. Each of up to 30 open files may have both an end file and an error trap, for a total of up to 60 local error trap destinations. They may be cleared, or altered, under program control, as often as necessary.

Global errors may relate to any function, and encompass local errors if a local error trap was not declared. Global error trapping is enabled by executing an ON ERROR GOTO statement. It may be cleared, or altered, under program control, as often as necessary.

When an error occurs, the following priority schedule is followed:

1- If the error is an end file error (error code 2), and an end file trap has been enabled for that particular file, then control is transferred to the defined label or line number.

2- If the error is a local error (error code of less than 16), and a local error trap has been enabled for that particular file, then control is transferred to the defined label or line number.

3- If the error occurred in an OPEN or CREATE statement, and a local ERROR option was included, then control is transferred to the defined label or line number, even if the error is not a local error (the error code number is greater than 15).

4- If global error trapping has been enabled (with an ON ERROR GOTO statement), then control is transferred to the defined label or line number.

5- If no error trapping is enabled, a descriptive message is displayed on the console, followed by the object code address of the statement causing the error, and program execution is terminated.

Syntax conventions for this manual

[]	optional item
{}	optional item which may be repeated
avar	array variable
cavar	control array variable
dest	line number or label
dr:	drive name (A: - P:)
expr	expression
lit	literal
lnum	line number
name	valid function name
navar	numeric array variable
nlit	numeric literal
nvar	numeric variable
relop	relational operator
savar	string array variable
slit	string literal
stmt	BASIC/Z statement
svar	string variable
var	variable
x,y,z	numeric expression
x$,y$,z$	string expression

APPENDIX A SYNTAX CONVENTIONS

ASCII Character Set - Collating Sequence

char		dec	hex	char	dec	hex	char	dec	hex	char	dec	hex
NULL	^@	00	00	SPACE	32	20	@	64	40	`	96	60
SOH	^A	01	01	!	33	21	A	65	41	a	97	61
STX	^B	02	02	"	34	22	B	66	42	b	98	62
ETX	^C	03	03	#	35	23	C	67	43	c	99	63
EOT	^D	04	04	$	36	24	D	68	44	d	100	64
ENQ	^E	05	05	%	37	25	E	69	45	e	101	65
ACK	^F	06	06	&	38	26	F	70	46	f	102	66
BEL	^G	07	07	'	39	27	G	71	47	g	103	67
BS	^H	08	08	(	40	28	H	72	48	h	104	68
HT	^I	09	09	)	41	29	I	73	49	i	105	69
LF	^J	10	0A	*	42	2A	J	74	4A	j	106	6A
VT	^K	11	0B	+	43	2B	K	75	4B	k	107	6B
FF	^L	12	0C	,	44	2C	L	76	4C	l	108	6C
CR	^M	13	0D	-	45	2D	M	77	4D	m	109	6D
SO	^N	14	0E	.	46	2E	N	78	4E	n	110	6E
SI	^O	15	0F	/	47	2F	O	79	4F	o	111	6F
DLE	^P	16	10	0	48	30	P	80	50	p	112	70
DC1	^Q	17	11	1	49	31	Q	81	51	q	113	71
DC2	^R	18	12	2	50	32	R	82	52	r	114	72
DC3	^S	19	13	3	51	33	S	83	53	s	115	73
DC4	^T	20	14	4	52	34	T	84	54	t	116	74
NAK	^U	21	15	5	53	35	U	85	55	u	117	75
SYN	^V	22	16	6	54	36	V	86	56	v	118	76
ETB	^W	23	17	7	55	37	W	87	57	w	119	77
CAN	^X	24	18	8	56	38	X	88	58	x	120	78
EM	^Y	25	19	9	57	39	Y	89	59	y	121	79
SUB	^Z	26	1A	:	58	3A	Z	90	5A	z	122	7A
ESC	^[	27	1B	;	59	3B	[	91	5B	{	123	7B
FS	^\	28	1C	<	60	3C	\	92	5C	¦	124	7C
GS	^]	29	1D	=	61	3D	]	93	5D	}	125	7D
RS	^^	30	1E	>	62	3E	^	94	5E	~	126	7E
US	^_	31	1F	?	63	3F	_	95	5F	DEL	127	7F

APPENDIX B ASCII CHARACTER SET

BASIC/Z Compiler Errors

During the compilation process, a problem with the source programs, system software, or system hardware may prevent completion. Should a compiler error occur, the problem will be reported, via one of the following messages, and the compile command will be aborted.

Array error in line xxxxx -
Invalid array reference.

Disk full -
BASIC/Z has run out of disk space while attempting to write either the .BZO or .PRN file.

Disk i/o error -
BASIC/Z was unable to read or write a disk sector.

Duplicate definition in line xxxxx -
Data item (array, function, label, etc.) was defined more than one time.

Fatal ??????? error in line xxxxx -
Any error message which begins with the word "Fatal" is an indication that BASIC/Z has located a system error within itself. Please contact System/z, inc. with specific details at once.

File not found -
Source file specified in the command, or in an INCLUDE statement, cannot be located.

Function error in line xxxxx -
Error was found in a function definition or reference, such as non-matching dummy arguments, illegal statement, etc.

Line number error in line xxxxx -
Line number referenced is not present in the program.

Memory overflow -
Not enough memory for the symbol table at compile time.

Memory overflow on definition of xxxxx -
Memory requirement exceeded 64k on a DIM statement.

Next w/o for in line xxxxx -
The NEXT statement cannot be matched with a FOR.

Object memory overflow error in line xxxxx -
Memory requirement exceeded 64k on compilation of code.

Overflow error in line xxxxx -
Program contains too many DO/UNTIL, WHEN/WHEND, or WHILE/WEND loops, or they are nested too deeply (limit: 255 each).

BASIC/Z Compiler Errors (continued)

Read only disk -
A disk was swapped without executing reset.

Read only file -
Previous version of the object file is marked read only, and therefore can't be scratched.

Select error -
Compile command specified an invalid drive name.

Sizes error in line xxxxx -
The sizes statement contains an invalid argument.

Source from incompatible revision -
Designated source file was created by a revision of BASIC/Z which is syntactically incompatible.

Subscript error in line xxxxx -
Array subscript is not valid.

Syntax error in line xxxxx -
Invalid syntax appeared in a program line.

Undefined reference error in line xxxxx -
A data item (array, function, label, etc.) was referenced without defining it.

Unmatched do in line xxxxx -
The DO statement cannot be matched with an UNTIL.

Unmatched until in line xxxxx -
The UNTIL statement cannot be matched with a DO.

Unmatched wend in line xxxxx -
The WEND statement cannot be matched with a WHILE.

Unmatched when in line xxxxx -
The WHEN statement cannot be matched with a WHEND.

Unmatched whend in line xxxxx -
The WHEND statement cannot be matched with a WHEN.

Unmatched while in line xxxxx -
The WHILE statement cannot be matched with a WEND.

BASIC/Z Run-time Errors

During execution of a compiled program, a problem with the program, user input, system software, or system hardware may cause generation of a run-time error. These may be termed either local errors, which have an error code number of less than 16, or global errors, which encompass all run-time errors. More complete information about error handling may be found in section 7 of this manual, or in section 4, under each reserved word related to the error handling process.

In the following table, the first column contains the error code number (returned by the function ERR), followed by the error message string (returned by the function ERR$). Lastly, a brief additional description of the error follows.

Local Error Codes

0 - No error	No error has occurred
1 - Disk i/o error	A disk sector could not be read or written
2 - End file	An attempt was made to read a record past the logical EOF position
3 - Disk full	Entire disk and/or directory is full
4 - File not found	Designated file does not exist
5 - Duplicate name	Attempt to create a file that already exists
6 - Parameter error	Specification of an invalid record #
7 - Lock error	Attempt to access a locked file or record
10 - Read only file	File can't be written to
11 - Read only disk	Disk can't be written to
12 - Select error	Invalid drive name for this system
13 - Reclen error	Invalid record length specification

Global Error Codes

16 - File not open	Specified file number is not open
17 - Invalid file name	Disk file name is invalid
18 - Invalid file number	File number outside the range of 0-29
19 - File open	File number specified in open or create is already open to another file
20 - Wrong file type	Attempt to access a sequential file with functions reserved to random, or vice-versa
21 - Memory overflow	Insufficient memory for execution
22 - Insufficient data	Not enough data items to complete a READ, GET, or INPUT statement
23 - Numeric overflow	Result of an operation is too large to be contained in a real variable
24 - Invalid argument	Argument to a function is not valid
25 - Array index error	Array element specified does not exist, or array not yet dimensioned
26 - Next w/o for	NEXT is executed without an associated FOR statement previously executed
27 - Invalid return	Return executed without previous gosub
28 - Stack overflow	Gosub 's nested beyond 128 levels
29 - Data type error	Wrong data type returned by an assembly language function call, or an attempt to an invalid numeric with INPUT or GET
30 - Chain error	Chain executed to a program without identical common or sizes
31 - Divide by zero	Division by zero attempted
32 - Log error	Negative or zero argument to LOG or LN
33 - Square root error	Negative argument to SQR
34 - Numeric range error	An attempt to assign a value to a numeric variable which is outside of its range
35 - DDIM error	Invalid index, duplicate DDIM, or memory overflow
36 - User function error	Illegal statement before FNEND executed

User Configuration Area

The user configuration area is a reserved block of memory which is used to store key information about the operating environment, as well as special data items which may be accessed by a compiled program. Any of the locations defined below may be interrogated through the use of a PEEK function. Also, a few locations may be altered with a POKE statement. However, it is absolutely crucial that no location be altered, unless it is specifically authorized in the following definitions.

The base address of the user configuration area is returned by the function SETUP. Each entry in the table below includes an offset value to specify its memory location relative to that base address, and they should be referenced in that manner. For example, the function PEEKWORD(SETUP+2) returns the BASIC/Z revision number, while PEEK(SETUP+18) returns the current user number.

We will make every effort to avoid changing offset values in future revisions of BASIC/Z. However, in all likelihood, some alterations will be inevitable. To lessen the impact at a later date, it may be prudent to assign offset values to variables at the start of each program, so that a single change will easily update all references. For example:

```
10 config.rev#  = setup +  2
20 config.user# = setup + 18
30 print "The user number is"; peek(config.user#)
```

In the following table, t/f means true/false (i.e. if the condition described is true, the memory location contains a non-zero value and if false, it contains zero).

Offset	Name	Bytes/Description
0	@BASE	2-operating system base address
2	@REVISION	2-BASIC/Z revision number, as a packed bcd word
4	@FORM^FLAG	1-t/f- printer recognizes form-feeds - this may be altered to force line-feeds for odd size pages
5	@LIST^COLS	1-printer maximum columns per line
17	@STREAM	1-current direction of the print stream 0=null 1=console 2=lprinter 4=spool
18	@USER	1-current user number
63	@CONS^LINES	1-console lines per page
64	@CONS^COLS	1-console columns per line
65	@ATTR^SGL	1-t/f- console limited to one video attribute
144	@CURSOR	1-t/f- cursor addressing supported
169	@CLEAR	1-t/f- clear screen supported
182	@ERAEOL	1-t/f- erase to end of line supported
189	@ERAEOS	1-t/f- erase to end of screen supported

Offset	Name	Bytes/Description
196	@BELL	1-t/f- console bell supported
203	@RVIDEO	1-t/f- reverse video supported
217	@BLINK	1-t/f- blinking video supported
231	@ULINE	1-t/f- underlined video supported
245	@LOWINT	1-t/f- low intensity supported
226	@HIGHINT	1-t/f- high intensity supported
290	@PWIDE	1-t/f- two wide printer characters supported
304	@PHIGH	1-t/f- two high printer characters supported
375	@ED^CHAR	1-fill character used to display the maximum field size for EDIT$ - this may be altered
376	@ED^INSERT	1-t/f- insert mode effective for the EDIT$ function - this may be altered
381	@ED^PROMPT	1-t/f- suppress display of the starting value and fill characters for EDIT$ - this may be altered
382	@ED^FINISH	1-t/f- suppress terminating erasure of fill characters and cursor movement to starting position for EDIT$ - this may be altered
383	@ED^FILL	1-t/f- suppress display of fill characters for EDIT$ - if true, @ED^CHAR must be set to a space (16R20) - this may be altered

PATCH Correction Utility

Due to the overlay structure of BASIC/Z, it is not possible to modify the supplied programs by conventional methods. However, be assured that System/z, inc. is committed to providing you with continuing, and ongoing support for BASIC/Z.

In the event that any "bugs" are discovered in BASIC/Z, an appropriate modification will be documented through publication of a System/z Software Information Bulletin. These will be provided, free of charge, to all users who have properly executed and returned their software license agreement.

When published, the Software Information Bulletin will contain complete operation instructions for PATCH, as the program will serve no useful purpose until that time.

The PATCH utility will function only with the System/z programs with which it is supplied. A separate version of PATCH will provided with any other program for which this function is required. Please make no attempt to use PATCH in another way, as serious damage could result.

APPENDIX E

PATCH

INSTALL Configuration Utility

The INSTALL Configuration Utility is provided to allow you to define your system requirements, as well as personal preferences, for the BASIC/Z compiler, the RUN/Z run-time library, and any command (.COM) files created with the BIND command. Prior to any regular use, each of these must be configured, through separate executions of INSTALL.

The INSTALL configuration utility consists of five (5) disk files:

INSTALL.BZO
INSTALL1.BZO
INSTALL2.BZO
INSTALL3.BZO
INSTALL.DAT

INSTALL.DAT is a data file which contains significant information needed by the installation programs. At the time of execution, it must reside on the same disk drive as the program files. INSTALL.BZO, a compiled BASIC/Z program, is the primary installation program, and the one which is actually executed. The remaining three program files may not be directly executed, but rather are called, as needed, by INSTALL. You may wish to note that INSTALL3.BZO is needed only when installing the BASIC/Z compiler. It is never accessed during an installation of the RUN/Z run-time library, or a command (.COM) file created with the BIND command.

As noted in section 1.013 of this manual, you may wish to use the BIND command to combine INSTALL.BZO with an unconfigured version of RZ.COM, into an executable command file named INSTALL.COM. While this offers added convenience to you, it is mandatory if you wish to distribute the INSTALL programs to third-party end users, as RZ.COM may not be distributed as a stand-alone program.

A word of caution - never, under any circumstances, allow the programs on your master disk to be configured with INSTALL. As originally supplied to you, they are pre-configured to execute properly with any supported operating system. However, should you install them, it is possible you might be unable to execute INSTALL again, if you change to a new operating system! For safety, always execute INSTALL with an unconfigured version of the run-time library (RZ.COM).

RUNNING INSTALL:

As INSTALL is a compiled BASIC/Z program, it is executed by typing:

RZ INSTALL <RETURN>

First, a "sign-on" message will be displayed, which includes the program name, serial number, copyright notice, etc. It will then begin to request pertinent information from you.

Throughout the installation process, you will notice that many questions require just a single character response. Others may need a variable length response of two or more characters, followed by a carriage-return, as a line terminator. To avoid ambiguity, whenever your response must be terminated by a carriage-return, the question will specifically instruct you to press <RETURN>.

On most installation questions, the program will display the current status of each item, before asking for your input. This will allow you to default to the current value, and change only specific items. This may prove particularly valuable in installation of a terminal which does not appear on our menu, but which is similar to one that is included. In this case, INSTALL would first be executed to choose the similar terminal. Upon completion, it would be executed again, but choosing no terminal from the menu this time. This non-standard configuration will cause INSTALL to display the current values for each individual special video function, allowing you to retain the correct ones, while altering others, as needed. Of course, this technique will also allow you to easily change individual attributes of supported terminals, should your preferences run counter to ours!

Due to the magnitude of BASIC/Z, the installation process is, of necessity, a rather lengthy one. However, we have attempted to make every question as self-explanatory as possible. With reasonable care, it can be completed with no further reference to this manual. For that reason, we will not attempt to duplicate the dialogue here, but just point out a few areas of special interest.

The initial question by INSTALL will allow you to choose to echo all installation dialogue to the printer, so that you may retain a copy for future reference. If you respond yes, it will next allow you to input a remark line, so that the listing may be noted with date/time, or any other important data.

A critical question is the name of the file to be installed. This could be the BASIC/Z compiler (BZ), the RUN/Z run-time library (RZ), or a command file which was previously created with the BIND command. Depending upon your choice, INSTALL will select different branches through the program, as different information is needed for each.

Another special area is the choice of operating system, as serious incompatibilities exist. For in-house use, the choice will be obvious. If you are pre-installing packages for third-party distribution, your choice will be dictated by the destination hardware. However, if you are supplying the INSTALL utility to a third party, be certain that the command file (created with BIND) utilized an unconfigured version of the run-time library. If there is any question of this, use choice #1 "Non-specific operating system", to return it to an unconfigured condition.

A further important point is the choice of terminal device. We support many popular systems, but obviously not every one. If you should find that your system is not supported as a standard device, you will be asked to input all control codes individually. While these should all be readily available in your hardware manuals, we are prepared to offer all possible assistance. Also, we would appreciate hearing of successful installations, to consider adding them as standard devices.

TR-MDOS Translator Utility

The TR-MDOS Translator Utility is provided to convert various files from a disk configured for the Micropolis Disk Operating System (MDOS) into a format usable by BASIC/Z on a CP/M format disk. To utilize this utility, your system must have the ability to access disks in the standard Micropolis MOD I or MOD II format. As TR-MDOS is a compiled BASIC/Z program, it is executed by typing:

RZ TR-MDOS <RETURN>

The following file types may be converted:

BASIC/Z SOURCE PROGRAMS :

Prior to conversion, the BASIC/Z source programs on the MDOS disk must be saved in the ascii mode, using the &A option to the SAVE or RESAVE commands. Due to the similarity of syntax, they are copied "as-is".

MICROPOLIS BASIC PROGRAMS :

The following conversion is automatic:

1- all tokens are expanded to the ascii equivalent
2- minor syntax changes are made (PLOADG=RUN, etc.)
3- delimiting spaces are inserted, as needed.
4- you may find extra spaces inserted in remarks and string literals, an unavoidable by-product of the Micropolis tokenization scheme

DATA FILES (type 0):

These are data files created by the Micropolis BASIC Interpreter, or BASIC/S compiler. They have a fixed record length of 250 bytes, and are converted to a random data file with a reclen of 250. Any record with a logical length less than 250 should be written with a trailing string delimiter, to avoid reading invalid data under CP/M. Each MDOS record has six random "garbage" bytes at the end, which are discarded.

DATA FILES (type 80):

These are data files created by the MDOS BASIC/Z compiler. They are random files which may have any logical record length, and are converted to random data files of the same record length.

DATA FILES (type 84):

These are sequential data files, identical in format to BASIC/Z for CP/M. They are therfore copied "as-is".

USER COMMENT FORM

We welcome your comments, suggestions, and your criticisms, as they help us to provide you with a better product.

Date _____ Title _____ Rev.# _____ Serial# _____

Did you find errors in either our software or documentation?
(Please be as specific as possible, including page numbers)

What is your opinion of this product? How could we improve the software or the documentation?

Name _____

Company _____

Address _____

City / State / ZIP _____

Telephone _____

Comments and suggestions become the property of System/z, inc.

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 36 / RICHTON PARK, IL

POSTAGE WILL BE PAID BY ADDRESSEE

System/z, inc.

P.O. Box 11
Richton Park, IL 60471

Attn: Systems Review

USER COMMENT FORM

We welcome your comments, suggestions, and your criticisms, as they help us to provide you with a better product.

Date _____ Title _____ Rev.# _____ Serial# _____

Did you find errors in either our software or documentation?
(Please be as specific as possible, including page numbers)

What is your opinion of this product? How could we improve the software or the documentation?

Name _____

Company _____

Address _____

City / State / ZIP _____

Telephone _____

Comments and suggestions become the property of System/z, inc.

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 36 / RICHTON PARK, IL

POSTAGE WILL BE PAID BY ADDRESSEE

System/z, inc.

P.O. Box 11
Richton Park, IL 60471

Attn: Systems Review

System/z

BASIC/Z

BASIC/Z

native code compiler

System/z,