

Interactive BASIC Compiler

by
Andrew R. Gariepy,
Scott Terry, David Overton,
Greg Branche and Halbert Liang

Documentation by
Michael A. Gariepy

© Copyright 1985, 1986, 1987



All Rights Reserved

Zbasic™ is a Trademark of Zedcor, Inc.

Fourth Edition: 4/87
First Edition: 8/85
Second Edition: 10/85
Third Edition: 5/86

TECHNICAL SUPPORT: 1-(602) 795-3996
Support hours: Monday-Friday, Noon to 5PM, Mountain Standard Time

Zedcor provides free phone support. Be sure to have your invoice number and license agreement number ready. You may need them to get support. Also be sure you understand the problem clearly and describe it as simply as possible. It is usually a good idea to test the problem a few times before calling. Sometimes it's just a syntax problem. Collect or toll free calls will not be accepted for technical support.

In addition, you may contact us on the **Genie Information Service** by sending electronic mail (EM AIL) to: ZBASIC. We check our mailbox semi-regularly and will respond to you via electronic mail. We also have topics set up on the various Round-Table Bulletin Boards for general information.

Notes on the Fourth Edition

This edition of the Zbasic™ manual contains all the computer appendices. This includes the appendix for MS-DOS™, APPLE™ //e, //c (DOS 3.3 and ProDOS), MACINTOSH™, CP/M™, and TRS-80™ Model 1, 3 and TRS-80 Model 4.

The appendices are at the back of the manual and the new index includes entries for both the reference section and the appendices. It is important to study the appendix for the computer you are using since there are usually enhancements and variations that are important to note.

Acknowledgements

Special thanks to John Kemeny and Thomas Kurtz for creating BASIC, the easiest and most powerful of all the general purpose languages. To Joanne Garipey for many late hours of editing. An extra special thanks to the programming teams that have meant so much to the success of the product. Scott Terry, Dave Overton, Greg Branche and Hal Liang and to Thomas Dimitri and David Cooper for their help with the MSDOS version. Special thanks to Karen Moesh and Leyla Blisard for making sure Zbasic gets mailed as fast as it does and to Apple Computer, Inc. for the Macintosh™, Laserwriter™, MacDraw, and MacPaint graphic software and to Microsoft for Word™; on which this entire manual was composed and printed (both text and graphics).

Many thanks to the multitudes of Zbasic™ users who provided helpful suggestions for this fourth edition.

Copyright Notice

The Zbasic software and manual are © Copyright 1985, 1986, 1987, Zedcor Inc., All Rights Reserved. It is against the law to copy the Zbasic software or Documentation on cassette, diskette, tape, listing or any other medium for any purpose other than the purchaser's archival use without the express written permission of Zedcor Inc.

Trademarks

Because Zbasic™ runs on so many different products, this manual makes reference to many manufacturers and products that are trademarks of their respective owners. We acknowledge these companies and thank those that have been so helpful in making Zbasic™ a success: Zbasic™ is a trademark of Zedcor, Inc. Apple, //GS, //e, //+, //c, Macintosh Plus, MAC XL, LISA and Macintosh™ are registered or licensed trademarks of Apple Computer, IBM PC, IBM PCjr., IBM PC-XT, PC-AT, and PC-DOS are registered trademarks of International Business Machines Corporation. MSBASIC, MS, Xenix, and MS-DOS are registered trademarks of Microsoft Corporation. CP/M is a registered trademark of Digital Research. Turbo Pascal is a registered trademark of Borland International. TRS-80, Radio Shack, Tandy 2000, Tandy and Tandy 1000, 1200, 3000 are registered trademarks of Tandy Corporation. Kaypro II, 4, 10, 16 and 286i are registered trademarks of Kaypro Corporation. Amiga, Commodore 64 and 128 are trademarks of Commodore International. Franklin Ace 1000 is a trademark of Franklin Corporation. Osborne is a trademark of Osborne Corporation. Compaq and Deskpro are trademarks of Compaq Computers. Panasonic Senior and Executive Partners are trademarks of Panasonic Industrial Corporation. Data General One is a trademark of Data General Corporation. Quadram is a trademark of Quadram Corporation. VAX is a trademark of Digital Equipment Corporation. Unix is a trademark of AT&T Corporation. We apologize for any unintentional omissions.

Zedcor Incorporated reserves the right to make changes to the specifications of Zbasic™ and to the Zbasic™ documentation without obligation to notify any persons or organization of such changes.

ZEDCOR, INC.
4500 East Speedway Blvd. Suite 22
Tucson, Arizona 85712-5305
(602) 795-3996
(602) 881-8101
Orders: 800- 482-4567

TABLE OF CONTENTS

Introduction	5
Getting Started	7
Configuration	11
Standard Line Editor	14
Adding, Inserting and Deleting Lines	14
The Line Editor	15
Renumber, Calculator, SAVE, LOAD, DIRectory	18
Running Zbasic Programs	22
RUN, RUN+, RUN*	23
Breaking Out of Executing Programs	24
Chaining Programs	26
Compile and Runtime Errors	28
Terms and Definitions	32
Math	36
Operators	36
Expression Evaluation types	37
Other Math Functions	40
Derived Math Functions	41
Order of Precedence	42
Conditional and Logical Operators	44
Numeric Conversions	46
Constants	48
Variables	50
Integer Variables	53
Floating Point Variables (REAL)	54
Configuring Accuracy	57
String Variables	61
Defining String Lengths with DIM and DEFLEN	64
INDEX\$	67
Arrays	72
Shell and Quick Sort examples	76
Graphics	110

TABLE OF CONTENTS

Files	98
Terms	100
File Structure	106
Sequential Method	108
Random Method	115
Mixing File Methods	120
Disk Errors	122
Screen and Printer	126
Keyboard Input	130
Loops FOR-NEXT, WHILE-WEND, DO-UNTIL	135
Functions and Subroutines	140
DEF FN	140
LONG FN	141
Machine Language	143
Program Structure	147
Debugging Tools	153
Porting Programs	156
Converting Old Programs	161
Keywords	168

Glossary	170
-----------------	-----

The reference section contains a complete alphabetical list of all Standard ZBasic commands, statements, functions and operators with cross reference to other commands and sections of the manual.

Also see the appropriate appendix for special commands or enhancements for a particular computer model.

TABLE OF CONTENTS

Computer Appendices

VERSION NOTES

Throughout this manual are notes to different versions of ZBasic.
An icon representing the various computer type is used.

Remember the icon for your computer type. If you see the icon
in the standard reference manual, a note will follow it describing
something of importance for that version.

MSDOS



IBM(r) PC, MSDOS(tm), PC-DOS(tm) and compatibles A

Z80



Z80(tm), TRS-80(tm) Model 1,3 and 4 and CP/M(tm)-80 B

Apple DOS 3.3

Apple ProDOS



Apple(r) IIe, IIc, Laser 128(tm) and //GS: DOS 3.3 C
Apple(r) II+, //e, //c Laser 128(tm) and //GS ProDOS D

Macintosh



Macintosh(tm), Macintosh Plus(tm), MAC XL(tm),
Macintosh SE(tm), Macintosh II(tm) and
LISA(tm) with Macworks(tm) E

INDEX

END

INTRODUCTION



As the original developer of ZBasic and the head of the programming team I want to thank you for your support.

I've been involved in writing Zbasic for eight years now and am very proud of what we've accomplished. It hasn't been easy but it's sure been fun. How many times does a complex product like ZBasic ever make it to market?

Over the years I have received thousands of suggestions from programmers. I've tried to implement as many of these suggestions as I could. I still need your feedback and comments so I can make ZBasic the most powerful programming tool available. Send your suggestions to the "ZBasic Wish-List Department" or to my attention.

Special thanks to my wife Janis for putting up with my programming late into the night and to the many ZBasic users that have taken the time to send letters of encouragement.

Andrew R. Gariepy
April, 1997

INTRODUCTION



ZBasic has come a long way since it was introduced in 1985. Many thousands of copies, on many different computers, have been distributed all over the planet.

We have accomplished what we set out to do; to provide a powerful, fast, interactive, simple-to-use, inexpensive BASIC compiler that works the same way on many different computers so you only have to learn a language once.

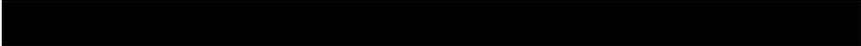
I've worked hard to make the manual simple to follow and easy to understand.

I highly recommend that you subscribe to the ZBasic newsletter; "Z". It covers all sorts of topics about ZBasic and has listings for public domain ZBasic subroutines on diskette you can get cheap. It's jammed with hints and tricks from other ZBasic users all over the world and from the ZBasic programmers themselves. Call 800-482-4567 to order.

Thank you for your support of ZBasic. Please let us know if you have any ideas of how to improve the product.

Michael A. Gariepy
April, 1987

GETTING STARTED



GETTING STARTED



GETTING STARTED

ZBasic is provided on a diskette for your computer. Before loading ZBasic do the following:

1. Read, sign and return the License agreement in the front of this manual. Keep track of your serial number, you may need it for support.
2. Read the Appendix for your computer. It will explain any variations or enhancements for your version of ZBasic and also has important information about hardware requirements or limitations.
3. MAKE A BACKUP COPY OF THE ORIGINAL ZBasic DISKETTE. *Never use the original diskette.* If you do not know how to make backups, refer to your DOS or User Manual.
4. Using the BACKUP, load ZBasic according to the instructions for your computer below:

MS-DOS	From A>:	ZBASIC
CP/M-80	From A>:	ZBASIC
TRS-80	From DOS READY:	ZBASIC
Apple DOS 3.3	From FP prompt:	BRUN ZBASIC
Apple ProDOS	From FP prompt:	- /ZBASIC/ZBASIC . SYSTEM
Macintosh	Using the mouse:	<i>Double Click</i> ZBasic Icon

HOW TO BE A ZBASIC EXPERT IN TEN MINUTES OR LESS

The following is a quick-and-dirty course that teaches you how to TYPE, RUN, SAVE, QUIT and LOAD a program using ZBasic.

First LOAD ZBasic according to the instructions for your computer above or in your computer appendix. Some versions require that you press <E> to enter the editor. If a prompt appears asking for input, press <E>. See CONFIGURE for more information about the options being offered.



Macintosh users note that the following lessons are done in the COMMAND window.

GETTING STARTED

LESSON ONE: TYPING IN A SIMPLE PROGRAM

When you see the message; **ZBasic Ready**, you may begin entering programs. So we may demonstrate the simplicity of ZBasic, please type in the following program exactly as shown. Always type **COMMANDS** in UPPERCASE and remember to press <ENTER> or <RETURN> at the end of each line.

```
10 FOR Count = 1 to 10
20 PRINT "Hi, I'm ZBasic!---"
30 NEXT Count
```

Congratulations, you've just entered your first ZBasic program. To see a listing of the program type: **LIST<ENTER>**. To find out more about entering and editing programs, see: **STANDARD LINE EDITOR**. Also see your computer appendix for information about using a full screen editor (if your version has one).

LESSON TWO: RUNNING THE PROGRAM

To run the program you just entered type:

```
RUN
```

The program will print the message; **Hi, I'm ZBasic!---** ten times. ZBasic actually compiles the program but does it so fast that you'll barely notice. When the program is finished you're back in the editor. That's the beauty of interactive compiling.

LESSON THREE: SAVING THE PROGRAM

To save your program, make sure you have an unprotected diskette in the drive and type:

```
SAVE MYPROG
```

The program will be saved to disk for future use.

LESSON FOUR: EXITING ZBASIC

To exit ZBasic type:

```
QUIT
```

You will now be back in the operating system. It's a good idea to save your programs before doing this.

LESSON FIVE: HOW TO LOAD EXISTING PROGRAMS

To load the previously saved program, first re-load ZBasic then type:

```
LOAD MYPROG
```

The program you saved is now back in memory. To see it, type **LIST**:

```
10 FOR Count = 1 to 10
20 PRINT "Hi, I'm ZBasic!---"
30 NEXT Count
```

GETTING STARTED



A NOTE TO EXPERIENCED BASIC PROGRAMMERS:

Since the ZBasic Compiler is very similar to the BASIC interpreters found on most microcomputers (except for graphic commands and file I/O), use the Reference Section and your Computer Appendix to check syntax differences from other BASIC's. Use the Index to find more in-depth answers. The appendices in the back of this manual contain the commands and enhancements for specific computers. These appendices are also very useful for converting programs from one machine to another.

If you have been frustrated with incredibly slow interpreters and awkward, complicated compilers, you will be pleased with the power and ease of ZBasic.



A NOTE TO INEXPERIENCED BASIC PROGRAMMERS

This manual is not intended to teach you BASIC programming from scratch. If you lack programming experience we suggest picking up some of the BASIC tutorials for the IBM PC, CP/M systems or the TRS-80, available from most major bookstores and libraries. Once you learn the beginning concepts of BASIC programming, like GOSUB, FOR/NEXT and that type of thing, this manual should be all you need.

ZBasic is very similar to the IBM PC, TRS-80, MSBASIC and GW BASIC interpreters; however, most Graphic commands and Random File commands are different (sequential file commands are very similar).

For those with some experience, this section and the section "Standard Line Editor" are written in a tutorial format.

Be sure to examine the appendix in the back of this manual for your computer. It will tell you about any differences and enhancements that are important to know before you start.

CONFIGURATION

CONFIGURATION



CONFIGURATION OPTIONS

Since no two programmers are alike, we allow you to configure your version of ZBasic. Most versions start with a screen something like this:

```
*****
* ZBasic (real version) BASIC Compiler *
* (c)1985, Redcar Inc. *
* August, 1985 *
*****
<E>dit
<C>onfigure
<S>ave
<P>atch (--Not all systems)

Enter Option: _
```

As you can see below, configuring your version of ZBasic is simple. Simply set the parameters the way you want, then save the reconfigured ZBasic:

- <E>**dit Type "E" to enter the Standard Line Editor. Once in the editor, you may LOAD, TYPE, RUN, EDIT, SAVE or DEBUG your programs.
- <C>**onfigure Typing "C" allows you to configure certain parts of ZBasic. Note that in most cases you will not have to change parameters. See next page for options.
- <S>**ave Typing "S" allows you to save ZBasic with the configuration defaults set to your options. This way you don't have to reconfigure ZBasic every time you load it.
- <P>**atch Type "P" allows you to make patches to ZBasic. If we make minor changes you won't have to return you disk to us for an upgrade. Not available on all versions.

CONFIGURATION

CHANGING CONFIGURATION

It is simple to change configurations. If the default value is not to your liking simply type in the value you want. Press <ENTER> to skip inputs, Press <BREAK> or <CNTR C> to go back to the main menu.

STANDARD CONFIGURE QUESTIONS	HEX	Decimal	INPUT
1. Double Precision Accuracy 6-54	000E	00014	?_
2. Single Precision Accuracy 2-52	0006	00006	?_
3. Scientific Precision 2-Double Prec.	0006	00006	?_
4. Maximum File Buffers Open 0 - 99	0002	00002	?_
5. Array Base 0 or 1	0000	00000	?_
6. Rounding Number 0 - 99	0031	00049	?_
7. Default Variable Type: <S>ingle, <D>ouble, <I>nteger		I	?_
8. Test Array Bounds <Y/N>		N	?_
9. Convert to uppercase <Y/N>		N	?_
10. *Optimize expressions as Integer? Y/N		Y	?_
11. *Space required after Keywords? Y/N		N	?_

* Not all versions.

DEFINITIONS OF THE STANDARD CONFIGURE QUESTIONS

1. Set from six to 54 digits of precision for Double Precision math. Defaults to 14.
2. Set from four up to two digits less than Double precision. Defaults to 6.
3. Digits of math precision for Scientific functions (ATN, COS etc.)
4. Set the number of files you want OPEN at one time. Up to 99. Two is the default.
5. Array Base 0 or 1. Set zero or one as ARRAY start. Zero is default.
6. Rounding Factor. Sets rounding for PRINT USING and other things.
7. Set variable default to Integer, Single or Double precision.
Press I, S or D key. Same as DEFDBL, DEFSNG, DEFINT A-Z.
8. Check the runtime program (object code) for array values going out of DIM bounds.
(Slows the program down but is very good for debugging purposes)
9. Tells ZBasic to convert all lowercase entries to UPPERCASE.
The variable "FRED" is the same as the variable "Fred" if this is done.
10. Two ways to evaluate expressions. Integer or Floating Point.
Defaults to integer for speed and size. Set to NO if you want defaults as real.
11. Forcing a space after keywords allows you to embed keywords in variables.



IMPORTANT NOTE: If you change configuration, make sure all CHAINED programs have EXACTLY THE SAME CONFIGURATION. Otherwise unpredictable results may occur.



Macintosh: Select the "Configure" menu item to change or save configuration options.
MSDOS and ProDOS versions of ZBasic have a CONFIG command that allows resetting the options from the Standard line editor. ***CP/M, Apple DOS 3.3** and **TRS-80** versions may not have the last two options offered. Check the appropriate appendix for specifics.

STANDARD LINE EDITOR



STANDARD EDITOR

ZBasic comes with a Standard Editor that works the same way on all computers. While most versions of ZBasic now come with a full screen editor which is easier and faster to use, the Standard Editor allows you to do quick-and-dirty editing and direct commands like an interpreter.

Learning the Standard Editor will allow you to jump from one version of ZBasic to another without having to re-learn the full screen editor for that particular machine.

ENTERING THE EDITOR

Load ZBasic. When the screen says: **ZBasic Ready** you have entered the ZBasic Interactive Programming Environment (a fancy name for the Standard Editor) and may enter programs and type direct commands.

The Standard Line Editor requires each line of a program to have a line number for editing and reference purposes (labels are available too.) Line numbers may range from 0-65534. Each line can be up to 250 characters long. To add a line, type a line number and the text, or use the AUTO command to have ZBasic assign line numbers automatically (some versions of ZBasic will allow you to enter programs without using line numbers. Check your appendix). If you are loading a program without line numbers, they will be added automatically. Line numbers are used for editing in the Standard Line Editor only.

Important Note: Always type keywords and commands in uppercase. Select "Convert to Uppercase" under Configure if you don't want to worry about it.



Important Note: This entire section deals with commands that are to be executed from the Standard Line Editor. If you are in the full screen editor you will need to switch to the Standard Editor. See your computer appendix for Specifics.



This section of the manual refers to the COMMAND window. Switching between the COMMAND and EDIT windows is accomplished with COMMAND E.

STANDARD LINE EDITOR

ENTERING AND DELETING LINES

Type in the following example. Enter it exactly as shown, as we will use this text to illustrate the use of the line editor. Remember to use <ENTER> at the end of each line. This is how ZBasic recognizes a line and stores it in memory:

```
10 THIS IS AN EXAMPLE OF ADDING A LINE
20 THIS IS THE SECOND LINE
30 THIS IS THE THIRD LINE
```

If you make a mistake use <BACKSP> or to delete it. If you <ENTER> a line incorrectly just type it over again. To see the complete program type LIST:

LISTING A PROGRAM

To list a line, or range of lines, use **LIST** or just **L**:

YOU TYPE

```
LIST or L
LIST "SUBROUTINE"
LIST "FRED"-
LIST 100-200
LLIST-100
LIST 100- or L100-
<period>
<UP ARROW>
<DOWN ARROW>
L+
LLIST+
L+-100
<SPACE>
</>
LIST*
```

ZBASIC RESPONDS

```
Lists the complete program to the screen
Lists the line with that label
List all lines after and including the line with the label "FRED"
Lists lines from 100-200
Lists lines up to 100 to printer
Lists lines from 100 on
Lists the last line listed or edited
Lists previous line (or plus <+> key)
Lists next line (or minus <-> key)
Lists program without line numbers
Lists to printer without line numbers
Lists up to line 100 without showing line numbers
Single steps long listings. <ENTER> continues listing
Lists PAGE of lines (10 lines) to screen
Some systems: Highlights keywords on screen while listing
```

DELETING LINES

Deleting lines is accomplished in a number of ways. Examples:

YOU TYPE

```
1000 <ENTER>
DEL 1000
DEL 10-50
DELETE 50
DELETE 50-
NEW
```

ZBASIC RESPONDS

```
Deletes line 1000
Delete line 1000
Delete lines 10 through 50
Delete line 50
Delete line 50 and all lines after
Delete the entire program Careful!
```

NOTE: Labels may be used in place of line numbers (except first example)

ADDING OR INSERTING A NEW PROGRAM LINE

Add or insert a line by typing in a new line number followed by text (be careful not to use the number of a line already being used unless you want to replace it). To insert a line between line 10 and line 20, assign a number such as 15 to the new line (or another number between 10 and 20). To add a line at the end of the program, assign the line a number greater than the largest line in the program.

STANDARD LINE EDITOR

HOW TO EDIT TEXT ON A LINE

The Standard Line editor is used to edit lines in a program and to give commands directly to the compiler. Deleting inserting, changing or adding new text is easy and fast.

EDIT ANYTHING ON A LINE... EVEN LINE NUMBERS!

Unlike most BASICs, ZBasic allows you to edit anything on a line, even the line number. When a line number is edited, ZBasic creates a new line with that line number. The old line will not be deleted or changed. Very handy for avoiding redundant typing.

The ZBasic line editor functions the same way on all versions of ZBasic. Here are ALL the line edit keys you need to remember:

STANDARD LINE EDITOR KEYS

CURSOR MOVEMENT

<SPACE> Move RIGHT

<BACKSP> Move LEFT

<S>earch for <letter>

DELETE TEXT

<D>elete one character

<K>ill, Delete up to <letter>

<H>ack to end of line

INSERT TEXT

<I>insert characters

e<X>tend line

<ESC>ape Insert mode

OTHER

<A>bort changes

<ENTER> Keep changes

<C>hange character under the cursor

<BREAK> Abort changes (CTRL C on some systems)

CURSOR ARROW keys are often used instead of <SPACE> and <BACKSP>.



Macintosh: <ESC>=<TAB>, <COMMAND Period>=<BREAK>. **MSDOS** and **Apple II:** Cursor keys=<SPACE> and <BACKSP>. Delete key also works as <BACKSP>. <CNTRL C>=<BREAK>. **MSDOS:** Insert key = <I>. **CP/M:** <CNTRL C>=<BREAK>. TRS-80: <SHIFT up-arrow>=<ESC>.

USING THE LINE EDITOR

The command to edit a line is "EDIT" (or just "E") followed by a line number (or label). If no line number is used, the last line LIST(ed) or EDIT(ed) will be assumed (<COMMA> without <ENTER> will also edit the current line).

"EDIT 20" and "E20" do the same thing.

The following page describes the simple commands used to edit the characters on the line.

STANDARD LINE EDITOR

LEARNING THE COMPLETE STANDARD LINE EDITOR IN 10 MINUTES OR LESS

LISTING THE LINE YOU ARE EDITING <L>

To see the complete line you are editing, and put the cursor at the beginning of the line, press the <L> key. Remember: Line editor commands do not require <ENTER>.

MOVING THE CURSOR ON THE LINE n <SPACE> <BACKSPACE>

To move the cursor back and forth on a line, use <SPACE> or <BACKSP> (some systems) (don't use <ENTER>). To move the cursor multiple positions, use a number first.

SEARCH FOR CHARACTER n <S>

To move the cursor to a specific character on a line quickly, use the <S> key, (SEARCH), followed by the character to find. To move the cursor from the "T" in "THIS" to the "L" in "EXAMPLE", just type <S> and <L>.

```
00010 THIS IS AN EXAMPLE OF ADDING A LINE
00010 THIS IS AN EXAMP_
```

CHANGE CHARACTER UNDER CURSOR n <C>

To change the character under the cursor, press <C> followed by the new character. To change five characters, press the <5> key first, the <C> key, then the five keys to replace the old characters.

ABORT (UNDO) CHANGES <A>

To undo changes press the <A> key. All changes, additions and deletions will be aborted.

DELETE CHARACTERS n <D>

To delete characters in a line use the <D> key. Pressing <D> will delete the character under the cursor. To delete five characters press <D> 5 times or press the <5> key and the <D> key.

ESCAPE PRESENT MODE <ESC>

To escape from INSERT, SEARCH, CHANGE, EXTEND or KILL modes, press <ESC>.

DELETE UP TO A SPECIFIC CHARACTER n <K>

To delete, or KILL, a range of characters from the cursor to a specified character, use the <K> key.

INSERT CHARACTERS <I>

To insert text in a line, position the cursor where insertion is desired. Press the <I> key. Type in text or <BACKSP> to erase text. Almost any key may be typed except <ESC>, <ENTER> or <BREAK>.

<ESC>ape exits the INSERT mode.

DELETE TO END OF LINE <H>

To delete all the characters from the cursor position to the end of the line, press the <H> key (Hack off the remainder of the line).

MOVE TO END OF LINE AND ADD <X>

To move the Cursor to the end of the line and enter the INSERT MODE, press the "X" key (For eXtend). <ESC> will return to the regular line editor mode.

EXIT THE LINE EDITOR <ENTER> or <BREAK>

<ENTER>: Exit the line edit mode and ACCEPT all changes and additions.

<BREAK>: To exit the line edit mode and IGNORE all changes and additions.

* n is a number. If you type 4D, four characters are deleted, n=nth occurrence or n times.

STANDARD LINE EDITOR

USING OTHER EDITORS OR WORD PROCESSORS

Most versions of ZBasic now come with a Full Screen Editor. Check your computer appendix to see if you have one for your version. If you choose, you may also edit ZBasic programs with a word processor or some other editor. You will need to save the ZBasic program in ASCII using the **SAVE*** or **SAVE+** commands before editing.

In order for ZBasic to load a text file it requires that:

Line lengths must be less that 250 characters
Every line must be followed by a Carriage Return

If the text file does not contain line numbers, ZBasic will assign line numbers to the program starting with one, in increments of one. Use RENUM to renumber a program. ASCII text takes longer to LOAD and SAVE.

RENUMBER PROGRAM LINES

ZBasic renumbers lines in a program using the RENUM command.
Format:

RENUM [[NEW LINE NUMBER][[, OLD START,][INCREMENT]]]

YOU TYPE

RENUM
RENUM 100,,5
RENUM 100,20,5
RENUM,,100

ZBASIC RESPONDS

Lines start with 10, Increments of 10
Lines start with 100, Increments of 5
Renumber From line 20, Start with 100, Increments of 5
Renumbers all lines by 100

THE CALCULATOR (DIRECT MODE)

ZBasic has a built in calculator. Use "?" or "PRINT" in front of a calculation to see the results. You may also convert number bases like HEX, Binary, Octal and Unsigned Integer. (See BASE CONVERSIONS) Examples:

YOU TYPE

PRINT 123.2*51.3
?SQR(92.1)
PRINT 3/2*6
?3./2*6
?320/.0001

ZBASIC RESPONDS

6320.16
9.5968745
6 (Calculated in INTEGER)
9 (Calculated in FLOATING POINT)
3200000

NOTE: Unless you have configured ZBasic to default to floating point, Integer is assumed. If configured for "Optimize expressions as Integer", use a decimal point in an expression to force the result of a calculation to be floating point (see CONFIGURE).

STANDARD LINE EDITOR

SAVE, LOAD, APPEND and MERGE

ZBASIC uses the LOAD and SAVE commands to load and save programs. Subroutines saved in ASCII without line numbers may be inserted in your program with APPEND. To SAVE in ASCII use "***". To SAVE in ASCII without line numbers use "+". Examples:

SAVE MYPROG	Saves in tokenized format.
SAVE CHECKERS 2	Saves tokenized to TRS-80 drive 2.
SAVE* MYPROG	Saves MYPROG in ASCII.
SAVE+ TEST	Saves TEST without line#'s in ASCII.
LOAD CHECKERS	Loads Checkers.
LOAD* CHECKERS	Loads Checkers but strips REMarks and Spaces.
MERGE MYPROG	Merges program MYPROG.
MERGE* MYPROG	Merges ASCII program, strips REM's and Spaces.
APPEND 2000 MYSUB	Loads non-line# ASCII subroutine, MYSUB, to line 2000.
APPEND* 50 SORT	Loads SORT to line 50 in increments of 1, strips all REM's and Spaces from the routine.

NOTE: Only non-line numbered ASCII programs may be APPENDED (SAVE+). Only line numbered programs may be merged (SAVE or SAVE*).

When LOAD(ing) programs without line numbers, ZBasic assumes the end-of-line is terminated with <CR>, <CRLF> or 250 characters, whichever comes first. Lines are assigned line numbers starting with one, in increments of one.

FILE DIRECTORY OR CATALOG

To see the names of files on the current storage device type DIR. Examples:

MS-DOS (also see PATH and CHDIR)

Apple DOS 3.3 and **CP/M:**

DIR	Lists all the files on the present drive
DIR B:	Lists the files on drive B
DIR A:	Lists all the files on drive A
DIR C:	Lists all the files on drive C

NOTE: The Apple DOS 3.3 version of ZBasic uses A, B, C... for drive specs instead of D1, D2...

APPLE ProDOS: (also see PATH)

DIR	Lists all files in current directory
DIR FRED	Lists all files in subdirectory FRED
DIR FRED/TOM	Lists all files in subdirectory TOM

TRSDOS:

DIR 0	Lists the files on drive zero
DIR 2	Lists the files on drive two
DIR 1	Lists the files on drive one

Macintosh: (also see FILES\$)

DIR HD30:Fred	Lists files in folder called "Fred" on root directory call HD30
LDIR HD30:Fred	Lists all files to the printer

Be sure to see your COMPUTER APPENDIX for variations.

STANDARD LINE EDITOR

THE MINI-COMPILER (Direct mode similar to an interpreter)

The Mini-compiler permits compilation of one line programs while in the standard editor. This is very convenient for testing logic or math without having to run the entire program. You are limited to one line but may use a colon ":" to divide a line into multiple statements.

Remember to use ? or PRINT to see the results. Examples:

<u>YOU TYPE</u>	<u>ZBASIC RESPONDS</u>
PRINT LEFT\$("HELLO",2)	HE
PRINT CHR\$(65)	A
PRINT ASC("A")	65
FOR X=1 TO 500: ? X::NEXT	1 2 3 4 5 ...500
? ABS(TAN(1)* EXP(2)+ LOG(9))	13.704997622614
: LPRINT "HELLO"	Prints "HELLO" to the printer
PLOT 0,0 TO 1024, 767	Plots a line across the screen
? &AB	171 (HEX to decimal)

*Note: A Mini-Compiler line may not start with an "E" or "L" since these are used for abbreviations for EDIT and LIST. To do a command that starts with "E" or "L", use a colon ":" first; :LPRINT

THE FIND COMMAND

ZBASIC will FIND variables, quoted strings, labels, line numbers and commands within a program quickly and easily. In most cases simply type FIND followed by the text you want to find. The only two exceptions are:

1. To find quoted strings, use one leading quote; FIND "HELLO"
Note 1: First characters in quoted string are significant.
Note 2: "A" and "a" are considered different characters.
2. Use "#" in front of a line number reference: FIND #1000

<u>YOU TYPE</u>	<u>ZBASIC FINDS</u>
FIND "HELLO"	01010 A=20:PRINT"HELLO THERE"
FIND A\$	01022 Z=1:A\$=B\$:PRINTA\$+B\$
or...	01333 ABA\$="goodbye"
FIND 99	05122 F=2:X=X+2+F/999
FIND #12345 (line number)	08000 GOTO 12345
FIND 100 (not a line number)	02000 X=100
FIND X(C)	03050 A=1:T=ABS(X(C)/9-293+F)
or...	03044 ZX(C)=4
FIND PRINT	00230 A=92:PRINTA
FIND "SUB5"	00345 "SUB500": CLS
or...	03744 GOSUB "SUB500"
FIND OPEN	03400 OPEN"R",1,"FILE54",23
FIND X=X+2	09922 F=2:X=X+2+F/999
FIND <ENTER>	Finds next occurrence
<> (semi-colon key)	Finds next occurrence

To FIND data in remarks or DATA statements use FIND REM ..., FIND DATA ...

Note: If your version of ZBasic comes with a full screen editor, you may have other FIND or REPLACE options. See your computer appendix for specifics.

STANDARD LINE EDITOR

SETTING CHARACTER WIDTH AND MARGINS FOR PROGRAM LISTINGS

ZBasic has powerful formatting commands for making program listings to the screen or printer easier to read.

WIDTH, WIDTH LPRINT AND PAGE

Since screen and printer widths vary depending on the hardware, the user may set the width of listing to either the printer or the screen.

COMMAND	RESULT
WIDTH=0 THROUGH 255	Sets Screen width for listings.
WIDTH LPRINT= 0 THROUGH 255	Sets the printer width for listings.
PAGE 0-255(1), 0-255(2), 0-255(3)	Formats LINES PER PAGE for printer. (1) Desired lines printed per page (2) Actual lines per page (3) Top Margin

An example of using these commands for printer listings: To set the top and bottom margins to 3 lines each (to skip perforations) and the printer width to 132, type:

WIDTH LPRINT=132: PAGE 60,66,3

NOTE: WIDTH, WIDTH LPRINT and PAGE may also be used from within a program. Check the reference section for specifics. (In a program, the PAGE function returns the last line printed. The PAGE statement will send a form feed to the printer. A ZERO value disables all the functions above.

AUTOMATIC LOOP AND STRUCTURE INDENTING

For readability, loops are automatically indented two spaces. When WIDTH is set, lines that wrap around will be aligned for readability as in line 10. Completed loops on the same line will show an asterisk at the beginning of the line as in line 120:

LIST+ (without line numbers)	LIST (with line numbers)
CLS: REM THIS IS A LONG	00010 CLS: REM THIS IS A LONG
STATEMENT THAT CONTINUES...	STATEMENT THAT CONTINUES...
FOR X= 1 TO 10	00020 FOR X= 1 TO 10
DO G=G+1	00025 DO G=G+1
GOSUB "Graphics"	00030 GOSUB "Graphics"
UNTIL G=3	00035 UNTIL G=3
NEXT	00040 NEXT
"MENU"	00050 "MENU"
CLS	00060 CLS
END	00070 END
"Graphics": X=0	00080 "Graphics": X=0
DO X=X+16	00090 DO X=X+16
PLOT X, 0 TO X, 767	00100 PLOT X, 0 TO X, 767
UNTIL X>1023	00115 UNTIL X>1023
*FOR X= 1 TO 100: NEXT	00120 *FOR X= 1 TO 100: NEXT
RETURN	00125 RETURN



Note: LLIST*+ may also be used to do program listings to the Imagewriter or Laserwriter without line numbers and with keywords highlighted as above.

RUNNING ZBASIC PROGRAMS



RUNNING ZBASIC PROGRAMS

There are a number of ways to compile your programs with ZBasic. The most commonly used is a simple RUN. This lets you compile and debug interactively. Definitions:

RUN COMPILE PROGRAM IN MEMORY AND EXECUTE

The interactive mode is the easiest and fastest way to write and debug your programs. In many ways it is similar to a BASIC interpreter since you may:

- 1. RUN a program to check for errors**
- 2. *BREAK out of a running program by pressing <BREAK>.**
- 3. Return to ZBasic to re-edit the program.**

Interactive compiling is limited to available memory. If a program gets too large you will have to use one of the methods below. ZBasic will tell you when this is necessary with an "Out of Memory" message.

RUN filename COMPILE PROGRAM FROM DISK AND RUN

If a program gets too large for interactive compiling using just RUN, the program text may be saved (not in ASCII), compiled, and executed. This is possible because the text to be compiled is no longer resident and frees up memory for the compiled program.

RUN* COMPILE PROGRAM IN MEMORY AND SAVE TO DISK

RUN* filename COMPILE FROM DISK AND SAVE TO DISK

Compiles the program from memory (RUN*) or disk (RUN* "filename") and saves it to disk. A few moments later ZBasic will request the filename of the resulting compiled program to be saved (For IBM or CP/M use a .COM suffix. For TRS-80 use a /CMD suffix).

This method frees up the most memory for the final program because the source code and ZBasic are no longer resident in memory. Compiled programs saved to disk are machine language programs and should be executed from the operating system like any other machine language program. See column three of the COMPILE MEMORY CHART.

RUN+ COMPILE PROGRAM IN MEMORY AND SAVE AS CHAIN PROGRAM

RUN+ filename COMPILE FROM DISK AND SAVE AS CHAIN

See CHAINING PROGRAMS for details.

RUNNING ZBASIC PROGRAMS

DETERMINING MEMORY REQUIREMENTS

MEM returns the available memory. (The table may vary on some versions).

TYPE	MEM:	MEANING
00123	Text	Program text memory used (source code).
49021	Memory	Free memory.
00000	Object	Compiled program size of object code.*
00000	Variable	Memory required for variables.*

*Type **MEM** immediately after compiling to get the correct totals. At other times the results of "Object and Variable" may be invalid.

TYPICAL MEMORY USAGE BY "RUN" TYPE

RUN

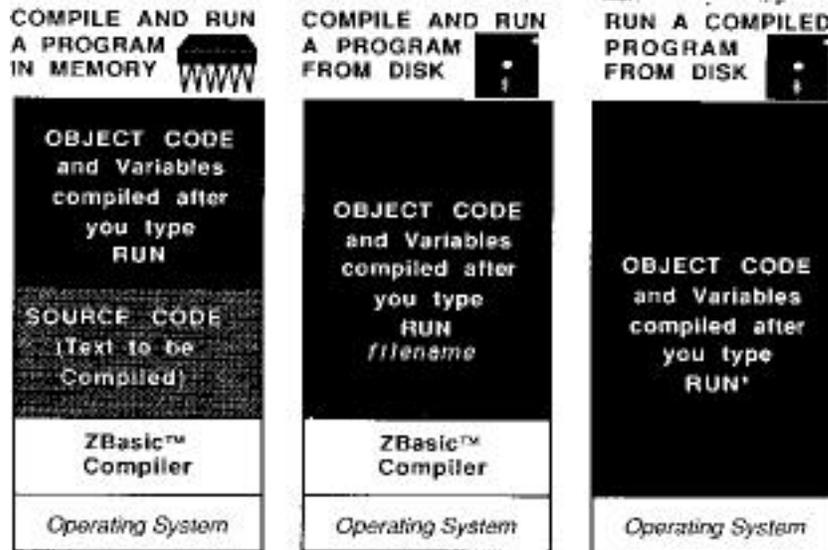
Program text is resident in memory with ZBasic, the compiled program and the variables used by that program. The user may press <BREAK> when running the program, re-enter

RUN filename

The program text is saved to disk and compiled from the disk to memory and RUN. Larger programs may be compiled this way because the program to be compiled is not in memory. the editor and debug any mistakes and re-compile.

RUN* [filename]

The program is compiled from memory or disk and the resulting machine language program is saved to disk. The program is executed as a machine language program. When this program is executed the program text and ZBasic are no longer resident, leaving more memory for the program.



*See your Computer Appendix to determine actual memory usage.

RUNNING ZBASIC PROGRAMS

<BREAK>ING OUT OF RUNNING PROGRAMS

To make a program STOP when the <BREAK> key is pressed, use TRON, TRONS, TRONB or TRONX.

TRONB	Checks at the start of every line to see if the <BREAK> key has been pressed. If pressed ZBasic returns control to DOS or to the Standard line editor (if in interactive mode). To disable TRONB use the TROFF command.
TRONS	Single step trace. CNTR Z to engage/disengage any other key to single step through the program a statement at a time.
TRON	Displays line numbers during runtime.
TRONX	Checks for the <BREAK> key at the beginning of that line only.

NOTE: TRONX, TRON, TRONS and TRONB may cause INKEY\$ to miss keys. TROFF turns all the TRON functions off. All TRONs will slow down programs AND increase size.

USING INKEY\$ TO SET BREAK POINTS

You may also use INKEY\$ to break out of a program. Put the following line in a program loop or wherever you may want to escape:

```
IF INKEY$="S" THEN STOP
```

Program will stop if the "S" key is pressed (any key could have been used).

CASES WHERE BREAK WILL NOT FUNCTION

Since ZBasic compiles your programs into machine language, there occurs certain situations where the <BREAK> key will be ignored. Remember; the <BREAK> key is checked only at the beginning of a line. The following example will not break:

```
TRONB
*FOR X= 1 TO 10: X=1: NEXT
```

This is obviously an endless loop (X never gets to 10). One obvious way around this is to avoid putting the entire loop construct on one line.

Examples of other cases where the <BREAK> key is ignored; INPUT, LINE INPUT, DELAY and SOUND statements.



Macintosh: <BREAK>=<COMMAND Period>. <CNTR Z>=<COMMAND <Z>. Most people use BREAK ON instead of TRONB with the Macintosh. See Appendix. **Apple II:** <BREAK> means: <CNTR C>, <CNTR RESET> may be preferable. **MSDOS:** <BREAK> means: <CNTR C>. **CP/M:** <BREAK> means: <CNTR C>. **TRS-80:** <BREAK> means the <BREAK> key.

CHAINING





CHAINING PROGRAMS TOGETHER

Chaining is convenient when programs are too large for memory and must be broken into smaller programs. There are three ways to chain programs:

- o CHAIN WITH SHARED VARIABLES (GLOBAL or COMMON VARIABLES)
- o CHAIN WITH INDEPENDENT VARIABLES
- o CHAIN WITH SOME VARIABLES COMMON AND OTHERS NOT



Macintosh CHAIN programs are limited to 28k. See "SEGMENT" and "SEGMENT RETURN" in the appendix for instructions on using the Macintosh memory manager.

EXAMPLES OF CHAINING PROGRAMS WITH SHARED VARIABLES

Programs that will share variables must have those variables defined in exactly the same order in all the programs being chained. ZBasic allows common or shared variables to be DEFINED within DIM statements (even if they are not arrays). CLEAR or CLEAR END should always be used to clear variables that are not shared. Examples:

"STARTB"

```
DIM A(10),100A$(100),Z,F5,W99
OPEN"I",1,"PROG1"
RUN 1
```

```
:REM Always execute this program 1st
:REM This is just a starter program
```

"CHAIN1"

```
REM THIS IS PROG1
TRONB: REM ENABLE <BREAK> KEY
DIM A(10),100A$(100),Z,F5,W99
CLEAR END
TV=23: PR=4
CLS: PRINT"THIS IS PROGRAM #1"
PRINT"Z=";Z,"F5=";F5
Z=RND(10):F5=RND(10)
PRINT"Z=";Z;" F5=";F5
PRINT"JUMPING TO PROGRAM #2"
DELAY 2000
OPEN"I",1,"PROG2"
RUN 1: REM RUNs Prog2
```

"CHAIN2"

```
REM THIS IS PROG2
TRONB
DIM A(10),100A$(100),Z,F5,W99
CLEAR END
ZZ=99: MYVAR=9191
PRINT "THIS IS PROGRAM #2"
PRINT"Z=";Z,"F5=";F5
Z=RND(10):F5=RND(10)
PRINT"Z=";Z;" F5=";F5
PRINT"JUMPING TO PROGRAM #1"
DELAY 2000
OPEN"I",1,"PROG1"
RUN 1:REM RUNs Prog1
```

CHAINING

COMPILING THE EXAMPLE PROGRAMS

1. RUN* STARTB and save as START
Always RUN* a START program. This is a dummy program and is used only to get the chained programs started and contains the runtime routines. Any filename will do.
2. RUN+ CHAIN1 and save as PROG1
3. RUN+ CHAIN2 and save as PROG2

NOTE: Always compile a START program using the RUN* command so that the chained programs have a runtime package. All chained programs must be compiled using RUN+.

USE "DIM" TO DEFINE SHARED OR COMMON VARIABLES

When chained together, both PROG1 and PROG2 will share variables defined on line 10 after the DIM. If F5 equals 10 in PROG1, it will still equal 10 when you RUN PROG2.

Because variables "TV" and "PR" are unique to PROG1 and the variables "ZZ" and "MYVAR" are unique to PROG2, CLEAR END must be used to initialize them (they must be assigned values). Otherwise false values will be passed from other CHAIN programs.

The example programs (PROG1 and PROG2) will chain back and forth until you press <BREAK>. Lines 80 and 90 are where the programs branch off to the other program.

CLEARING NON-SHARED VARIABLES WHEN CHAINING

Always use CLEAR END to clear variables that are not common between the programs. All variables that follow a CLEAR END will be unique to that program and will start out as null values.

(1)	(2)
10 DIM 200A\$(100), 65B\$(300)	10 DIM 200A\$(100), 65B\$(300)
20 CLEAR END	20 CLEAR END
30 DIM FR(900)	30 A9=10: Z=33

In the above examples, the array variables A\$ and B\$ are shared and will contain the same values, while all other variables in the program following the CLEAR END statement will be null or zero and unique to that program. FR(n) is unique to program (1) and A9 and Z are unique to program (2).

This statement may be used in non-chained programs as well. It is a handy way to null or zero out selected variables (the variables still exist, they are just set to zero or null).

CHAINING PROGRAMS WITHOUT SHARING VARIABLES

This is done exactly as the same as the previous examples for shared variables, except CLEAR is used on the first line of each chained program.

In the example programs CHAIN1 and CHAIN2, add a line:

```
3 CLEAR
```

Variables are not shared and CLEAR clears all variables (sets them to zero or null) each time a program is entered or chained.

To selectively share some variables and not others use the CLEAR END statement described on the previous page and in the reference section.



ERRORS

There are different types of error messages. When errors are encountered during compilation, compiling is stopped and the offending line is displayed. This is a Compile Time error. Errors encountered during execution of a program are called Runtime Errors.

COMPILE TIME ERRORS

After typing RUN, ZBASIC compiles the program. If errors are encountered, ZBASIC will stop compiling and display the error on the screen along with the offending line (when compiling from disk using RUN "Filename" or RUN*, ZBasic will stop compiling, load the Source Code, and LIST the line where the error occurred.) The Statement within the line and the line number will be displayed. The following program would cause ZBASIC to print an error during compile:

```
00010 CLS
00020 PRINT "HELLO THERE MR. COMPUTER USER!"
00030 PRINT "I AM A COMPUTER"
00040 Z=Z+1: X=X+Z: PWINTX
```

RUN

```
Syntax Error in Stmt 03 at Line 00040
00040 Z=Z+1: X=X+Z: PWINT X
```

NOTE: The error will be marked in some way depending on the computer system being used. The error marker indicates the general error location on the line where compilation stopped. To edit line 40 above type: EDIT 40 (or just comma). Fix the spelling of PRINT.

ZBasic will often display the missing character it expected.

```
00010 INPUT"Enter a number" A$
RUN
";" expected error in Stmt 01 at line 00010
00010 INPUT"Enter a number"_A$

00010 DIM A(10,10)
00020 A(X)=100
RUN
", " expected error in Stmt 01 at line 00020
00020 A(X_)
```

ERRORS

COMPILE TIME ERROR MESSAGES

A compile time error is one that ZBasic encounters after you type RUN (while it is compiling your program). More often than not, the error is a syntax error. Edit the line to fix the error and type RUN again until all the errors have been deleted.

COMPILE TIME ERROR MESSAGE

DIM Error in Stmt...

DEFINITIONS and POSSIBLE REMEDIES

Only constants may be used in DIM statements:

DIM A(X) or Z(A+4) are not allowed. If you have a need to erase and reuse dynamic starting arrays see: INDEX\$, CLEAR INDEX\$, MEM.

No DIM Error in ...

Array variable being used was not Dimmed. Make sure variable is Dimmed correctly. Most interpreters allow ten elements of an array before and DIM is required. A compiler requires a DIM for every array.

Overflow Error in ...

DEF LEN or DIM string length is less than one or greater than 255. Also if CLEAR =zero or CLEAR is too large. Check and adjust range.

Syntax Error in ...

Anything ZBasic does not understand. Check for spelling, formatting errors and syntax. The offending part of the line is often highlighted.

Too Complex Error...

String function is too complex to compile. Break up complex strings.

Re-DEF Error...

An FN or LONG FN was defined twice.

Variable Error in...

String assignment problem: A\$=123:Change to A\$=STR\$(123)

Out of Memory Error in...

Program is getting too large. Check large DIM statements and defined string lengths, or compile using RUN*. For very large programs you may wish to CHAIN programs together.

Line # Error in...

GOTO, GOSUB, ON GOTO, ON GOSUB, THEN or some other branching command can't find line number or a label.

Mismatch error in...

The assignment to a variable is the wrong type.

Structure Error in...

FOR without NEXT, DO without UNTIL, WHILE without WEND, LONG IF without END IF or LONG FN without and END FN.

Structure Error in 65535*

Missing NEXT, WEND, END IF, END FN, or UNTIL. If unable to find error quickly, LLIST the program. structures are indented two spaces. backtrack from the end of the program until the extra indentation is located.

"?" Expected error in ...

ZBasic expected some form of punctuation that was not provided. Check cursor position in displayed line for error.

*NOTE: Each ZBasic loop command must have one, and only one, matching partner. Each FOR need a NEXT, each WHILE needs a WEND, each LONG FN needs and End FN, each LONG IF needs an END IF and each DO needs an UNTIL.

RUN TIME ERRORS

A Run Time (execution) error is an error that occurs when the compiled program is running (Object Code). The only Run Time error messages produced are:

DISK ERRORS (Unless trapped by the user). See Disk Errors in the FILES section of this manual.

OUT OF MEMORY ERROR when loading a compiled program saved to disk that is too large to execute in memory.

ARRAY BOUNDS ERROR will be shown if the user configures ZBasic to check for this. This will slow down a program execution but is extremely handy during the debug phase of programming. You may turn this off after the program is completely tested. If access to an array element out of bounds is made, the program is stopped and the line number with the error printed.

STRING LENGTH ERROR. Some versions of ZBasic have a configure option that tells ZBasic to check for string assignments greater than the length allowed. This does slow execution speed and add memory overhead, so you may want to remove this error checking after the program is debugged. See your appendix for specifics. If an attempt is made to assign a string a value longer than its length, the program is stopped and the line number with the error is printed.

RECOVERING FORM FATAL RUNTIME ERRORS

Since ZBasic is a compiler and converts your code into machine language, there is always a risk that you may unintentionally enter an endless loop or hang up the system (the computer will not respond to anything).

In these instances you may not be able to get a response form the computer or be able to <BREAK> out o f the program. The system may have to be reset or turned off, and back on again to regain control. To avoid losing valuable time, it is very important that you **SAVE PROGRAMS** and **MAKE BACKUPS FREQUENTLY**. See you computer appendix for possible alternatives.

USING SINGLE STEP DEBUGGING TO FIND THE SOURCE OF "CRASHES"

Should you encounter a situation where your program goes so far and then the system hangs-up or you get a system error of some kind that you just can't locate, there is a simple way to find the problem.

First put a TRONS and TRON in the program somewhere before the crash occurs. The TRON is added so that you can see a listing of the line numbers as the program executes. Press the space bar a statement at a time, keeping track of the line numbers as they go by.

When the system crashes, make a note of the line number where the crash occurred and fix the problem in your program.

TERMS AND DEFINITIONS



TERMS AND DEFINITIONS

TERMS AND DEFINITIONS

I use terms throughout this manual that may be unknown to you. The following terms are used to make reading the technical information easier.



IMPORTANT NOTE

"The Hand" is pointing out something of importance for that section. Read it!

OPTIONAL

Items [enclosed in brackets] are OPTIONAL. You may or may not include that part of a command, function or statement.

REPETITION

Three periods (ellipsis) mean repetition ... when they appear after the second occurrence of something.

PUNCTUATION

Any punctuation such as commas, periods, colons and semi-colons included in definitions, other than brackets or periods described above, must be included as shown. Any text in Courier font, like this: COURIER FONT TEXT, means it is something for you to type in or a simulation of the way it will look on your screen like a program listing.

COMPUTER APPENDIX

Refers to the appendix in the back of this manual, ABOUT YOUR COMPUTER.

SPECIAL 32

The superscripted 32 means this command, function or statement only works on 32 bit computers. See you COMPUTER APPENDIX to see if your computer supports 32 bits. In this edition of the manual it refers to the Macintosh computer only.

ABBREVIATIONS

Frequently used line editor commands have convenient abbreviations:

USE WITH <ENTER>

?	PRINT
DEL	DELETE
E	EDIT
L	LIST

USE WITHOUT <ENTER>

,comma	EDIT present line
.period	LIST present line
/slash	LIST next 10 lines
;(semi-colon)	FIND next occurrence

TERMS AND DEFINITIONS

DIFFERENT (KEY) STROKES FOR DIFFERENT FOLKS

Since ZBASIC operates on many different computers, reference is made to the same keys throughout this manual.

<u>MANUAL USES</u>	<u>YOUR COMPUTER MAY USE</u>
<SPACE>	SPACE BAR
<BACKSP>	BACKSPACE, DELETE, LEFT ARROW
<BREAK>	CONTROL C, COMMAND PERIOD
<ENTER>	RETURN, CARRIAGE RETURN
<ESC>	ESCAPE, CNTRL UP ARROW, TAB
<UP ARROW>	CURSOR UP, PLUS KEY<+>
<DOWN ARROW>	CURSOR DOWN, MINUS KEY<->
<letter>	Press the Key with that letter

See your COMPUTER APPENDIX for variations or enhancements.

LABELS ON LINES

A line may have a label directly following the line number consisting of upper or lowercase, alphanumeric characters, or symbols in any order enclosed in quotes. The length of a label is limited to the length of a line. ZBasic recognizes only the first occurrence of a label.

Line numbers are essential only for line EDIT(ing), MERGE, and APPEND. Statements like; LIST, EDIT APPEND, GOTO, ON GOTO, GOSUB, ON GOSUB, DEL, etc., may use either Labels or line numbers. List programs without line numbers by using LIST+.

SIMPLE STRINGS

Quoted strings: "Hello", "This is within quotes"

Any String variables: A\$, NAME\$, FF\$, BF\$(23).

Any of the following string functions:

MKI\$, MKB\$, CHR\$, HEX\$, OCT\$, BIN\$, UNS\$, STR\$, ERRMSG\$, INKEY\$, INDEX\$(9).

COMPLEX STRINGS

Complex strings are any combination of SIMPLE STRINGS. Any string operations containing one of the following commands: simple string + simple string, LEFT\$, RIGHT\$, MID\$, STRING\$, SPACE\$, UCASE\$

ZBasic allows only one level of COMPLEX STRING expression. Complex strings MAY NOT be used with IF THEN statements. Convert all multi-level complex strings to simple strings:

CHANGE COMPLEX STRINGS

```
B$=RIGHT$(A$+C$,2)
B$=UCASE$(LEFT$(A$,3))
IF LEFT$(B$,2)="IT"THEN 99
```

TO SIMPLE STRINGS

```
B$=A$+C$: B$=RIGHT$(B$,2)
B$LEFT$(A$,3): B$=UCASE$(B$)
D$=LEFT$(B$,2): IFD$="IT"THEN 99
```



The Macintosh version allows much deeper levels of complex strings.

TERMS AND DEFINITIONS

VARIABLE TYPES

A\$, A#, A!, A%, and A%(n,n) represent different variables. If no type is given, integer is assumed (unless configured differently by the user or changed with DEF DBL, DEF SNG or DEF STR). A and A% would be the same variable. Types:

%	Integer variable
&	4 byte Integer (32 bit machines only)
!	Single precision variable
#	Double precision variable
\$	String variable

EXPRESSIONS



Throughout this manual reference is made to expressions. There are different types of expressions and the following words will be used to refer to specific expressions.

DEFINITION OF EXPRESSION

EXPRESSION refers to a combination of constants, variables, relational operators or math operators in either integer, floating point or string used to yield a numeric result. The following UNDERLINED examples are EXPRESSIONS.

CLEAR 2000

A= T+1

TEST= X^ 2.23* 5+1

IF X*3.4 <= Y*98.3 THEN Z= 45*84^R

IF A\$>B\$ AND B\$<>C\$ THEN GOTO 1000

TERMS AND DEFINITIONS

BYTE EXPRESSION

A **BYTE EXPRESSION** always results in a number from 0 to 255. The expression may be floating point, integer or string, but if the actual result is more than 255 or less than 0, the final result will return the positive one byte remainder. ZBasic will not return an error if the calculation result is out of this range.

INTEGER EXPRESSION

An **INTEGER EXPRESSION** results in an integer number from -32768 to 32767. The expression may be floating point, integer or string, but if the actual result is more than 32767 or less than -32768, the final result will return the integer remainder which is incorrect. ZBasic will not return an error if the calculation result is out of integer range.

Note: 32 bit computers have a LongInteger range of +-2,147,483,647.

UNSIGNED INTEGER EXPRESSION

An **UNSIGNED INTEGER EXPRESSION** always results in an unsigned integer number from 0 to 65535. The expression may be floating point or integer but if the actual result is more than 65535 or less than 0 the final result will return the remainder which will be incorrect. See UNS\$ for displaying signed integers as unsigned.

Note: 32 bit computers have an unsigned LongInteger range of 0 to 4,294,967,300.

CONDITIONAL EXPRESSION

Conditional expressions like $A=B$, $A>B$, $A<B$ etc., will return negative one if TRUE(-1), and zero (0) if FALSE.

It should be noted that a condition like IF X THEN... would be TRUE if X is non-zero and FALSE if X=zero.



IMPORTANT NOTE ABOUT MATH EXPRESSIONS: If you have configured numeric expressions to be optimized as integer, the final result of an expression will be evaluated by ZBasic as integer UNLESS one of the following conditions is found within that expression:

- * Constant with a type of (#, !, or exponent: D or E)
- * Constant with a decimal point (period). Example: .34 or 1.92
- * Non-integer variable. (Single or Double precision #, !)
- * MATH Functions: COS, SIN, ATN, SQR, LOG, EXP, TAN, VAL, CVB, FRAC, AND FIX.
- * Floating point math symbols \, ^ or [

Note: One expression may be made up of other expressions within parentheses. Each expression is evaluated separately and must meet the criteria above.



MATH OPERATORS

- + ADDITION
- SUBTRACTION
- * MULTIPLY
- / DIVIDE
- \ DIVIDE (Floating point Divide or Integer Divide)*
* If configured as "Optimize Expressions as Integer" the \ is forced floating point divide, otherwise it is forced integer divide.
- ^ or [EXPONENTIATION (raise to the power)
- MOD REMAINDER OF INTEGER DIVIDE (MODulo)
- << SHIFT LEFT (BASE2 MULTIPLY)
- >> SHIFT RIGHT (BASE2 DIVIDE)

NEGATION

Negation will reverse the sign of an expression, variable or constant. Examples: -A, -12, -.32, -(X*B+3^7), -ABS(Z*R)

SHIFT (binary multiply and divide)

Since computers do internal calculations in binary (BASE 2), SHIFT is used to take advantage of this computer strength. Multiply or divide SHIFTS are faster than floating point multiply or divide and may be used when speed is a factor. (Integer Shift Right loses sign). A good example; ATN(1)<<2 = pi (instead of the slower; ATN(1)*4)

- >>n Shift right (Divide by 2^n)
- <<n Shift left (Multiply by 2^n)
(n Truncates to an integer number)

SHIFT FUNCTIONS	BASE 2 Equivalent*	DECIMAL Equivalent	RESULT
4>>1 (Divide)	4/2^1	4/2	2
4<<1 (Multiply)	4*2^1	4*2	8
89.34<<2	89.34*2^2	89.34*4	357.36
.008>>1	.008/2^1	.008/2	4E-3
999.>>7	999/2^7	999/128	7.8046875

*2^1=2, 2^3 is the same as 2*2*2, 2^7 is the same as 2*2*2*2*2*2*2
With 10>>>8.231 or 10<<<8.231 the 8.231 would be converted to integer 8

MATH

REGULAR MATH EXPRESSIONS AND ZBASIC EQUIVALENTS

Regular math and algebraic expressions are quite similar to ZBasic expressions. The user should, however, be aware of some important differences. As in regular algebraic expressions, parentheses determine the part of the expression that is to be completed first. Examples:

Regular Math	ZBasic™ Equivalent
$A-2B+1$	$A-2^*B+1$
$A(\frac{C}{B})$	$A^*(C/B)$
$(A-B)+T^2$	$A-B+T^2$
$(AC)H^2$	$(A^*C)^(H^2)$
$(A+\frac{B^2}{C})T^6$	$(A+B^2/C)^*T^6$
$A(-B)$	A^*-B

FORCING EXPRESSION EVALUATION TO DEFAULT TO FLOATING POINT

ZBasic normally optimizes expression evaluation by assuming integer if no floating point types are seen in the expression. This can cause confusion for those used to MSBASIC or other languages without this capability. Setting "OPTIMIZE EXPRESSIONS FOR INTEGER MATH?" to "NO" sets the expression evaluator to interpret expressions as most other computer languages do; that is, all expressions will default to floating point if parentheses or any part of the expressions contain a floating point operator. While this makes it easier to follow the logic in an expression, the speed of execution time will suffer greatly.

It should be noted that a compiler cannot determine if an expression like $C\%=A\%*B\%$ returns a floating point number. If $A\%=20000$ and $B\%=20000$ an overflow will occur.

NOTE: Some versions of ZBasic, most notably versions older than 4.0, will not allow you to configure the expression evaluator. Older versions default to optimized integer math as described below.

WHY OPTIMIZE EXPRESSIONS FOR INTEGER MATH?

ZBasic defaults to a unique way of interpreting math expressions. Under CONFIGURE, you are given the option of setting expression evaluation to optimized integer or regular floating point. The default is INTEGER. This requires some extra thought on the part of the user but forces programs to execute much faster and much more efficiently.

UNDERSTANDING EXPRESSIONS THAT ARE OPTIMIZED FOR INTEGER MATH

Optimized Integer Expressions return the final result of an expression in integer or floating point, depending on how the expression is evaluated.

To optimize program speed and size, *integer is assumed UNLESS one of the following is found in an expression: decimal Point, scientific function, \ (floating point divide: SEE NEXT PAGE DEFINITIONS OF DIVIDE SYMBOLS) , #, ! or a constant >65,535.

The following examples will give you an idea how ZBasic evaluates expressions as Integer or floating point. (B=10)

<u>EXPRESSION</u>	<u>RESULT</u>	<u>EXPRESSION EVALUATED AS</u>
B* .123	1.23	FLOATING POINT (Decimal point force REAL)
B* 23	230	INTEGER
B *23#	230	FLOATING POINT (# forces Double Precision)
B* 32000	-11264	INTEGER (Overflow error)
B* 32000.	320000	FLOATING POINT (Decimal point)
SIN(B)	-.54402111	FLOATING POINT (Scientific Function)
B*0+65535	-1	INTEGER (UNS\$(-1)=65535)
B*4800	-17536	INTEGER (UNS\$(-17536)=48000)

*Note: You may configure ZBasic to assume floating point by setting "Optimize expressions for integer math" to "NO". See "Configure" in the beginning of this manual.

PARENTHESES IN OPTIMIZED INTEGER EXPRESSION EVALUATION

Parentheses are used to force an expression to be evaluated in a certain order. (See ORDER OF PRECEDENCE)

ZBasic evaluates an expression by examining the outermost portions. In the expression: X*(2*(4.03+4))*5, the innermost portion of 4.03+4 is floating point, but since the outermost portions of X* and *5 are integer the whole expression is returned as an integer. (B=10 in examples)

<u>EXPRESSION</u>	<u>RESULT</u>	<u>EXPRESSION EVALUATED AS</u>
B*(32000+1)	-7670	INTEGER (Out of range error)
B*(32000.+1)+0!	320010	FLOATING POINT (! forces REAL)
B+ (.23)+1200	1210	INTEGER
B+ (.23)+1200.	1210.23	FLOATING POINT (period forces REAL)
B+(200*(.001^2))	10	INTEGER
B+200*.001^2	10.0002	FLOATING POINT
B+ATN(2)	11.107149	FLOATING POINT (Scientific Function)

The expression within each level of parentheses is still evaluated according to the precision in that level.

NOTE: Newer versions of ZBasic may be configured to expression evaluation you are more used to. See "OPTIMIZE EXPRESSIONS FOR INTEGER MATH" above.

MATH

INTEGER AND FLOATING POINT DIVIDE SYMBOLS

It should be noted that the Divide symbols / and \ take on different meanings depending on the type of expression evaluation being used:

Optimized for Integer "YES"	Optimized of Integer "NO"
/ = Integer Divide	/ = Floating Point Divide
\ = Floating Point divide	\ = Integer Divide

SCIENTIFIC FUNCTIONS

ZBasic offers several scientific and trigonometric math functions for making many calculations easier.

SQR(expression) SQUARE ROOT of expression.
Returns the number multiplied by itself that equals expression. SQR(9)=3

LOG(expression) Natural LOGARITHM if expression (sometimes referred to as LN(n)).
Common LOG10 =LOG(n)/LOG(10)

EXP(expression) Natural logarithm base value:
e=2.718281828459045235602874135266249775724
TO THE POWER of EXPRESSION. Inverse of LOG.

LOG and EXP may speed up calculations dramatically in certain situations. Some comparative equalities using LOG and EXP:

$$\begin{aligned} X*Y &= \text{EXP}(\text{LOG}(X) + \text{LOG}(Y)) \\ X/Y &= \text{EXP}(\text{LOG}(X) - \text{LOG}(Y)) \\ X^Y &= \text{EXP}(\text{LOG}(X) * Y) \end{aligned}$$

CONFIGURING SCIENTIFIC ACCURACY

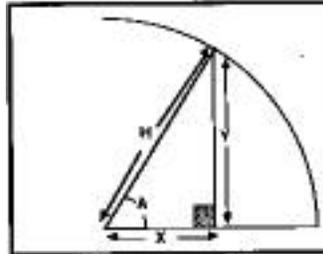
Scientific function accuracy may be configured up to 54 digits of accuracy (32 bit machines may be higher). Default accuracy is 6 digits. Scientific accuracy may be configured from two digits of accuracy, up to Double Precision accuracy (not necessarily the same as Single or Double precision).

Precision is set when loading ZBasic under <C>onfigure. Scientific math functions are complicated; the more digits of precision used, the longer the processing time required. See "Setting Accuracy" in the floating point section of this manual for information about accuracy, speed charts and memory requirements.

SCIENTIFIC MATH SPEED

When speed is more important than accuracy, configure DIGITS OF PRECISION (under configure at start-up) to 6 digits for DOUBLE, 4 digits for SINGLE and 6 digits for SCIENTIFIC.

TRIGONOMETRIC FUNCTIONS



- TAN**(expr) TANGENT of expression in radians.
 $TAN(A)=Y/X$, $X=Y/TAN(A)$, $Y=TAN(A)*X$
- ATN**(expr) ARCTANGENT of the expression in radians.
 $A=ATN(Y/X)$, $Pi=ATN(1)<<2$
- COS**(expr) COSINE of the expression in radians.
 $COS(A)=X/H$, $H*COS(A)=X$, $X/COS(A)=H$
- SIN**(expr) SINE of the expression in radians.
 $SIN(A)=Y/H$, $Y=H*SIN(A)$, $H=Y/SIN(A)$
- SQR**(expr) SQUARE ROOT of expression.
 $H=SQR(X*X+Y*Y)$

TAN, ATN, COS AND SIN return results in Radians.

OTHER ZBASIC MATH FUNCTIONS

- FRAC**(expr) Returns FRACTIONAL portion of an expression
 $FRAC(23.232)=-.232$, $FRAC(-1.23)=-.23$
- INT**(expr) Returns expression as a whole number
 $INT(3.5)=3$, $INT(99231.2)+0=99231$
- SGN**(expr) Returns the SIGN of an expression
 $SGN(-23)=-1$, $SGN(990)=1$, $SGN(0)=0$
- ABS**(expr) Returns the ABSOLUTE VALUE of an expression
 $ABS(-15)=15$, $ABS(152)=152$, $ABS(0)=0$
- FIX**(expr) Returns the whole number of an expression
 $FIX(99999.23)=99999$, $FIX(122.6231)=122$
 (Like INT but forces floating point mode)
- expr **MOD** expr Returns the remainder of an integer divide (MODulo)
 $9 \text{ MOD } 2=1$, $10 \text{ MOD } 2=0$, $20 \text{ MOD } 6=2$
- RND**(expr) Returns a random number between 1 and expr
 $RND(10)$ randomly returns 1,2,3,4...10
- MAYBE** Randomly returns -1 or 0. (50-50 chance)
 IF MAYBE PRINT "HEADS" ELSE PRINT "TAILS"

MATH

DERIVED MATH FUNCTIONS

<u>MATH FUNCTION</u>	<u>TERM</u>	<u>ZBasic EQUIVALENT EQUATION</u>
PI	() PI	ATN(1)<<2 (accurate to double precision)
e	e	EXP(1)
Common LOG 10	LOG	LOG(X)/LOG(10)
Area of a CIRCLE	R^2	Y#=(ATN(1)<<2)*Radius*Radius
Area of a SQUARE		Y#=Length*Width
Volume of a RECTANGLE		Y#=Length*Width*Height
Volume of a CUBE		Y#=Length*length*length
Volume of a CYLINDER		Y#=(ATN(1)<<2)*Height*Radius*Radius
Volume of a CONE		Y#=(ATN(1)<<2)*Height*Radius*Radius/3
Volume of a SPHERE		Y#=(ATN(1)<<2)*Radius*Radius*Radius*4/3
SECANT	SEC(X)	Y#=1/COS(X)
COSECANT	CSC(X)	Y#=1/SIN(X)
COTANGENT	COT(X)	Y#=1/TAN(X)
Inverse SINE	ARCSIN(X)	Y#ATN(X/SQR(1-X*X))
Inverse COSINE	ARCCOS(X)	Y#ATN(1*2-ATN(X/SQR(1-X*X)))
Inverse COSECANT	ARCCSC(X)	Y#ATN(1/SQR(X*X-1))+(X<0)*(ATN(1)<<2)
Inverse COTANGENT	ARCCOT(X)	Y#=ATN(1)*2-ATN(X)
Hyperbolic Sine	SINH(X)	Y#(EXP(X)-EXP(-X))/2.
Hyperbolic Cosine	COSH(X)	Y#=(EXP(X)+EXP(-X))/2.
Hyperbolic Tangent	TANH(X)	Y#=(EXP(X)-EXP(-X))/(XP(X)+EXP(-X))
Hyperbolic Secant	SECH(X)	Y#=2./(EXP(X)+EXP(-X))
Hyperbolic Cosecant	CSCH(X)	Y#=2./(EXP(X)-EXP(-X))
Hyperbolic Cotangent	COTH(X)	Y#=(EXP(X)+EXP(-X))/(EXP(X)-EXP(-X))
Inverse Hyperbolic Sine	ARCSINH(X)	Y#=LOG(X+SQR(X*X+1))
Inverse Hyperbolic Cosine	ARCCOSH(X)	Y#=LOG(X+SQR(X*X-1))
Inverse Hyperbolic Tangent	ARCTANH(X)	Y#=LOG((1+X)/(1-X))/2
Inverse Hyperbolic Secant	ARCSECH(X)	Y#=LOG((1+SQR(1-X*X))/X)
Inverse Hyperbolic Cosecant	ARCCSCH(X)	Y#=LOG((1-SGN(X)*SQR(1+X*X))/X)
Inverse Hyperbolic Cotangent	ARCCOTH(X)	Y#=LOG((X+1)/(X-1))/2
Derivative of LN(X) (Natural LOG)		Y3=1/X
Derivative of SIN(X)		Y#=-COS(X)
Derivative of TAN(X)		Y#=1+TAN(X)^2
Derivative of COT(X)		Y#=-1+(1/TAN(X)^2))
Derivative of ARCSIN(X)		Y#=-SQR(1-X*X)
Derivative of ARCCOS(X)		Y#=-SQR(1-X*X)
Derivative of ARCTAN(X)		Y#=1/(1+X*X)
Derivative of ARCCOT(X)		Y#=-1/(X*X+1)
Derivative of ARCSEC(X)		Y#=1/(X*SQR(X*X-1))
Derivative of ARCCSC(X)		Y#=-1/(X*SQR(X*X-1))
Derivative of ARCSINH(X)		Y#=1/SQR(1+X*X)
Derivative of ARCCOSH(X)		Y#=-1/SQR(X*X-1)
Derivative of ARCTANH(X)		Y#=1/(1-X*X)
Derivative of ARCCOTH(X)		Y#=-1/(X*X-1)
Derivative of ARCSECH(X)		Y#=-1/(X*SQR(1-X*X))
Derivative of ARCCOSSECH(X)		Y#=-1/(SQR(1+X*X))

See DEF FN and LONG FN for adding these math functions to your programs.

ORDER OF PRECEDENCE

In order to determine which part of a math expression is done first an order of precedence is used. The following math operators are performed in the this order.

1. (((1st)2nd)3rd)	Innermost expressions within parentheses always performed first
2. -	Negation (not subtraction)
3. NOT	Logical operator
4. ^ or [Exponential
5. *,/, \, MOD	Multiply, Divide, Floating point Divide, MODULO
6. +, -	Addition, Subtraction
7. =, >=, =, <=, =, <, >, <, <>, ><	Conditional operators
>>, <<	Shifts
8. AND, OR, XOR	Logical operator

ZBasic will calculate each operation of an expression in order of precedence, as defined by the table above. The final result of an expression depends on the order of operations.

If there are items of equal precedence in a n expression, ZBasic will perform those operations from left to right.

$$A\#=2+5-3*6+1/4$$

This expression is performed in the following order;

1. $3*6$
2. $1/4$
3. $2+5$
4. $(2+5) - (3*6)$
5. $(2+5-(3*6)) + (1/4.)$

$$A\#=-10.75$$



Important Note: If expressions are optimized for Integer Math, the decimal point after the 4 forces the result of the expression to be floating point. If the decimal point had been omitted, the result would be -11. See CONFIGURE.

MATH

USING PARENTHESES TO FORCE PRECEDENCE

Parentheses are used in math expressions to force ZBasic to calculate that part of an expression first. If a math operation is enclosed in parentheses, which in turn is enclosed within parentheses, the innermost expression will be calculated first.

$$A\#=2+5-3*6+1/4$$

To force the 2+5-3 part of the above equation to be calculated first, and then multiply that by 6 and add 1 second, with division by 4 last, you would express the equation like this:

$$A\#=((2+5-3) * 6+1) / 4.$$

The order of operations in this expression would be:

1. (2+5-3)
2. (2+5-3)*6+1
3. ((2+5-3)*6+1)/4.

$$A\#=6.25$$



Note: If Expressions are optimized for Integer Math; the outermost expression is used by ZBasic to determine whether the final result will be returned as integer or floating point.

The decimal point after the 4 forces the expression to be calculated as floating point (although each expression within parentheses is evaluated as floating point or integer depending on the rules of expressions). If the decimal point had been omitted the result would have been 6.

To use the standard rules of expression evaluation, set "Optimize Expression evaluation to Integer" to NO under configure. Math expressions will be done in the usual manner if this is done.

CONDITIONAL OPERATORS

The conditional operators return:

0 (zero)	If the Comparison is FALSE
-1 (negative one)	If the Comparison is TRUE
A non-zero expression	Is always TRUE
A zero expression	Is always FALSE

These symbols are used for comparing expressions and conditions.

=	Equal To
<>, ><	Not Equal To
<	Less Than
>	Greater Than
>=, =>	Greater Than OR Equal To
<=, =<	Less Than OR Equal To

Examples: (A\$"HELLO" AND A%=2000)

CONDITIONAL EXPRESSION	RESULT
X=12<20	X=-1
PRINT 23=45	0
IF 10>5 THEN PRINT "SURE IS"	SURE IS
IF A%-2000>100-99 PRINT A%	Nothing
IF VAL(A\$)=0 THEN PRINT A\$	HELLO
PRINT 2>5, 3<5, 5>5	0 -1 0
IF A%>120 THEN PRINT "OK"	OK
IF A%*5>=10000 THEN STOP	Program STOPs
IF A% PRINT "YES"	YES (Non zero is True)
PRINT 50>50	0
PRINT 50>=50	-1
IF A%>30000 THEN PRINT "OK"	Nothing
X=1: IF X THEN PRINT "YEP"	YEP
X=0: IF X THEN PRINT "YEP"	Nothing
X=77.321>77.320+1	0
X= "HELLO"="HELLO"	X=-1
IF A\$="HELLO" PRINT "YES"	YES
IF A\$="HELLO" PRINT "YES"	Nothing
IF A\$>"HEL" THEN PRINT A\$	HELLO
IF A\$<>"GOON" THEN PRINT "NO"	NO
IF STR\$(A%)=" 2000" PRINT "YES"	YES

MATH

LOGICAL OPERATORS

Zbasic makes use of the logical operators AND, OR, NOT, SHIFTS and XOR. These operators are used for comparing two 16 bit conditions and binary operations (except on 32 bit computers which can compare 32 bits). When used in comparative operations a negative one (-1) is returned for TRUE, and a zero (0) is returned for FALSE.

Logical Operators

condition **AND** condition
 condition **OR** condition
 condition **XOR** condition
 condition **SHIFT** condition
NOT condition

RETURNS

TRUE(-1) if both conditions TRUE, else FALSE(0)
 TRUE(-1) if either or both is TRUE, else FALSE(0)
 TRUE(-1) if only one condition is TRUE, else FALSE(0)
 TRUE(-1) if any non-zero value returned, else FALSE(0)
 TRUE(-1) if condition FALSE, else FALSE(0) if TRUE

EQV (emulate with)

NOT (condition XOR condition) TRUE(-1) if both conditions FALSE or both conditions TRUE, else FALSE(0)

IMP (emulate with)

(NOT condition) OR condition FALSE(0) if first condition TRUE and second condition FALSE, else TRUE(-1)

AND

1 AND 1 = 1
 0 AND 1 = 0
 1 AND 0 = 0
 0 AND 0 = 0

BOOLEAN "16 BIT" LOGIC

	00000001		00000111
AND	00001111	AND	00001111
=	00000001	=	00000111

OR

1 OR 1 = 1
 0 OR 1 = 1
 1 OR 0 = 1
 0 OR 0 = 0

	00000001		10000101
OR	00001111	OR	10000111
=	00001111	=	10000111

XOR

1 XOR 1 = 0
 0 XOR 1 = 1
 1 XOR 0 = 1
 0 XOR 0 = 0

	00000001		10000101
XOR	00001111	OR	10000111
=	00001110	=	00000010

SHIFT >>, <<

255 >> 2 = 63
 23 << 3 = 184

	11111111		00010111
>>	00000010	<<	00000011
=	00111111	=	10111000

NOT

NOT 1 = 0
 NOT 0 = 1

NOT	11001100	NOT	01111011
=	00110011	=	10000100



With the Macintosh, 32 bit integers may also be used with logical operators (LongInteger&).

NUMERIC CONVERSIONS



INTEGER BASE and SIGN CONVERSIONS

ZBasic has functions for converting integer constants to hexadecimal (BASE 16), octal (BASE 8), binary (BASE 2), unsigned integer and back to decimal (BASE 10). UNS\$, HEX\$, OCT\$ and BIN\$ are the functions used to convert an integer to the string representation of that SIGN or BASE.

DECIMAL TO BASE CONVERSION

<u>HEX</u>	<u>OCTAL</u>	<u>BINARY</u>
HEX\$(48964) ="BF44"	OCT\$(54386) ="152162"	BIN\$(255) ="0000000011111111"
HEX\$(32) ="0020"	OCT\$(8) ="000010"	BIN\$(512) ="0000000100000000"

BASE TO DECIMAL CONVERSION

<u>HEX</u>	<u>OCTAL</u>	<u>BINARY</u>
VAL("&0030") =48	VAL("&O000011") =9	VAL("&X0000000001100011") = 99

DISPLAYING UNSIGNED INTEGERS

To display or print an unsigned integer number use UNS\$. UNS\$ returns the unsigned value of the number by not using the leftmost bit as a sign indicator:
UNS\$(-1)=65,535, UNS\$(-2311)=63,225

ZBasic interprets the integers, -1 and 65,535 as the same value. In BINARY format they are both 1111111111111111. The left-most bit sets the sign of the number to positive or negative. This is the same unsigned integer format used by many other languages.



The same holds true with LongIntegers, only 32 bits are used instead of 16 bits. The signed range is +- 2,147,483,647. The unsigned range is 0 to 4,294,967,293. See DEFSTR LONG in the appendix for ways of using 32 bit HEX\$, OCT\$, UNS\$ and BIN\$.

NUMERIC CONVERSIONS

CONVERSION BETWEEN DIFFERENT VARIABLE TYPES

ZBasic will convert variables from one type to another as long as the conversion is within the range of the target variable.

DOUBLE or SINGLE PRECISION VARIABLE =INTEGER VARIABLE will convert exactly (unless single precision is set less than 6 digits).

INTEGER VARIABLE=DOUBLE or SINGLE PRECISION VARIABLE will convert correctly if the double or single precision variables are within the integer range of -32,768 to 32,767 (unsigned 0 to 65,535). Any fractional part of the number will be truncated. Results outside integer range will be the rounded integer result, which is incorrect, and no error will be generated.

SINGLE PRECISION VARIABLE=DOUBLE PRECISION VARIABLE conversions will be exact to the number of significant digits set for single precision since the calculations are done in double precision. If the single precision default is 6 digits and double precision is 14 digits, the 14 digit number would be rounded down to 6 digits in this example (precision is configurable by the user).

STRING VARIABLE=STR\$(INTEGER, DOUBLE OR SINGLE PRECISION VARIABLE) will convert exactly. The first character of the string produced is used for holding the sign. If the number is positive or zero, the first character of the string produced will be a SPACE, otherwise the first character of the string will be a minus (-).

INTEGER VARIABLE=VAL(STRING VARIABLE) will convert correctly, up to the first non-numeric character, if the string variable represents a number in integer range. Fractional portions will be ignored. Zero will be returned if not convertible.

DOUBLE OR SINGLE PRECISION VARIABLE=VAL(STRING VARIABLE) will convert correctly within the range of floating point precision set by the user (rounding will occur if it is more digits than the set precision).



LongInteger conversions are the same as regular integers with the exception that the range is much larger. Since all internal integer calculations are done in LongInteger, conversions are simple. See DEFSTR LONG in the Macintosh appendix.



CONSTANTS

Constants are values used in expressions, variable assignments, or conditionals. In the following underlined program lines, the constants values remain constant, while values of A\$, Z and T are variable.

```
10 PRINT"HELLO THERE": PRINT A$: Z=Z+T+2322.12
```

ZBasic users both string (alphanumeric) and numeric constants.

INTEGER CONSTANTS

An integer constant is in the range of -32,768 to 32,767 (or unsigned integer in the range of 0 to 65,535).

The BASE of an integer may be represented in Decimal, Hexadecimal, Octal or Binary. See "Numeric Conversions" for information about converting integers to and from HEX, OCTAL, BINARY and DECIMAL.

MEMORY REQUIRED FOR INTEGER CONSTANTS

Two bytes each in the same format as integer variables.



The Macintosh also has LongInteger constants with a range of +-2,147,483,647. LongInteger constants require four bytes memory each. Macintosh format of integer is the opposite of other versions. i.e. MSB is first and LSB is last.

CONSTANTS

FLOATING POINT CONSTANTS

The range of floating point constants is +-1.0E-64 to +-9.999E+63*. Constants may be expressed in scientific notation and/or up to 54 digits of significant accuracy.

Floating point constants are significant up to the double precision accuracy set by the user. If the number of digits is greater than the accuracy of double precision, it will be rounded to that precision. If the double precision default of 14 digits is assumed, a constant of 1234567890.123456 will be rounded to 1234567890.12345.

Constants may be forced as double or single precision by including a decimal point in the constant or by using # for double precision or ! for single precision.

MEMORY REQUIRED FOR FLOATING POINT CONSTANTS

ZBasic will store floating point constants in Binary Coded Decimal format (See Floating point variables memory requirements). This is based on the actual memory requirement of each constant, with a minimum memory requirement of 3 bytes per constant. To calculate the memory requirements of a specific constant use the formula:

NUMBER of DIGITS in the constant/2+1=Bytes needed*

Minimum of 3 bytes required per Floating point constant.



*the range of Double precision constants is E+-16,383 (single precision remains the same for compatibility). To calculate the memory required use the following equation ; Number of Digits/2+2=bytes needed (single precision is the same as above).



Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

STRING CONSTANTS

String constants are alphanumeric information enclosed in double quotes with the number of characters limited by line length (255 characters maximum).

```
"This is a string of characters"  
"12345 etc."  
"Hello there Fred"
```

Any character except quotes may be included between the quotes. To include quotes in string constants use CHR\$(34). PRINT CHR\$(34);"HELLO";CHR\$(34) would print: "HELLO". To conserve memory when using many string constants see PSTR\$.

MEMORY REQUIRED FOR STRING CONSTANTS

One byte plus the number of characters, including spaces, within the string constant. See PSTR\$ for ways of conserving memory with string constants.



VARIABLES

The word VARIABLE describes the label used to represent alterable values. ZBasic differentiates between four types of variables.

<u>VARIABLE TYPE</u>	<u>TYPE OF STORAGE</u>	<u>RANGE</u>
STRING	ALPHANUMERIC	0 TO 255 CHARACTERS
INTEGER	INTEGER NUMBERS	+32,767
SINGLE PRECISION	FLOATING POINT NUMBERS	E+- 63
DOUBLE PRECISION	FLOATING POINT NUMBERS	E+- 63



In addition to the variable types described above this version also supports LongInteger and an extended double precision range (single precision is the same as above).

LONG INTEGER	FOUR BYTE INTEGER	+2,147,483,647
DOUBLE PRECISION	FLOATING POINT NUMBERS	E+-16,383

Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

VARIABLE TYPE DECLARATION

Variable names may be followed by a type symbol:

\$	STRING VARIABLE
%	INTEGER VARIABLE
!	SINGLE PRECISION VARIABLE
#	DOUBLE PRECISION VARIABLE

If type is not given, integer is assumed (unless configured differently). A, A!, A\$, A#, A(2,2), A#(2,2), A!(2,2) and A\$(2,2) are considered different variables. Note: A and A% are the same variable if ZBasic is configured to Integer.



Type declaration for LongInteger is; &

VARIABLES

DEFINING VARIABLE TYPES

If you want to define variables beginning with a specific letter to be a specific type, use the DEF statement at the beginning of a program.

DEFSTR A-M,Z	Defines all variables starting with A thru M and Z as string variables. M and M\$ are the same variable.
DEFSNG A-C	Defines all variables starting with A thru C as single precision variables. C and C! are the same variable.
DEFDBL F,W	Defines all variables starting with F and W as Double precision variables. F and F# are the same.
DEFINT A,G,T-W	Defines all variables starting with A,G and T thru W as integer variables. No % needed. A and A% are considered the same variable.

Note: Even if a range of letters is defined as a certain type, a declaration symbol will still force it to be that type. For instance, if A-Z are defined as integer using DEFINT, A\$ is still considered a string, and A# is still considered a double precision variable.



DEFDBL INT A-M Defines variables starting with A thru M as LongIntegers. No & needed. A and A& are the same variable.

VARIABLE NAMES

Variable names must have the following characteristics:

- o Variable names may be up to 240 characters in length but only the first 15 characters are recognized as a unique variable.
- o First character must be in the alpha range of A-Z, or a-z.
- o Additional characters are optional and may be alphanumeric or underline.
- o Symbols not allowed: ",^/+->=<][()?" etc.

SPACE REQUIRED AFTER KEYWORDS

Many versions of ZBasic have this as a configure option. See "Configure". If you don't want to worry about embedding keywords in variables, set "Space Required after Keywords" option to "yes". It will require that keywords be followed by spaces or non-variable symbols. This allows variable names like FORD or TOM.

If you do not set this parameter, or do not have this option for your version of ZBasic, you must not embed keywords in variables.

UPPER/LOWERCASE WITH VARIABLES

If you want the variable TOM and the variable tom to be the same variable, you must configure "Convert to Uppercase" to "yes". See "Configure".

If you do not set this parameter, or do not have this option for your version of ZBasic, you must match case when using variables. i.e. TOM and tom are different variables.

VARIABLES

MEMORY REQUIRED FOR VARIABLES

<u>VARIABLES</u>	<u>MEMORY REQUIRED</u>
INTEGER %	2 bytes
STRING \$	256 bytes (default). String variable length is definable from 1 to 255 characters (plus one for length byte).
SINGLE PRECISION !	4 bytes (default)
DOUBLE PRECISION #	8 bytes (default) If Single or Double precision digits of precision is changed, use this equation to calculate memory requirements: DIGITS of ACCURACY /2+1=BYTES REQUIRED*

ARRAY VARIABLES

<u>ARRAY VARIABLES</u>	<u>MEMORY REQUIRED PER ELEMENT</u>
INTEGER %	2 bytes per element
STRING \$	256 bytes (default) per element. String variable length is definable from 1 to 255 characters per element. Add one byte per element to the defined length of the string for the length byte. DEFLen 200=201 bytes required per element.
INDEX\$(element)	1 byte plus the number of characters in an element
SINGLE PRECISION !	4 bytes (default) per element
DOUBLE PRECISION #	8 bytes (default) per element If FLOATING POINT digits of precision are changed, use this equation to calculate memory requirements: NUMBER OF DIGITS/2+1=BYTES REQUIRED*

Note: Remember to count the zero element if BASE zero is used.



Important Note: Some versions of ZBasic offer a high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.



*LongInteger variables and arrays use four bytes each. To determine double precision memory requirements for the Macintosh version: DIGITS/2+2=BYTES REQUIRED per variable or per double precision array element.

VARIABLES

INTEGER VARIABLES

Because ZBasic always attempts to optimize execution size and speed, it will always assume a variable is integer unless the variable is followed by a type declaration (#, !, \$, &) or that range of letters has been defined DEFSTR, DEFDBL, DEFDBL INT or DEFSTR. Although it will slow down program performance, you may force ZBasic to assume floating point variables under configuration. See "Configure". **Integer calculations may be 100 to 200 times faster than floating point!**

INTEGER RANGE

-32,768 to +32767



LongInteger range is +-2,147,483,647. Speed is as fast as regular integers.

DEFINING VARIABLES AS INTEGER

ZBasic assumes all unDEFined variables, or variables without type declarations (#,!,\$,&), are integer (unless configured differently by the user).

DEFINT may be used to force a range of variables starting with a certain letter to be integer with the DEFINT statement followed by a list of characters. For example: DEFINT A-G defines all variables starting with A,B,C...G to be integer. (G and G% would be the same in this case.)

To force a specific variable to be integer, even if that letter type has been DEF(ined) differently, follow a variable with %. TEST%, A% and F9% are integer variables.

INTEGER OVERFLOW RESULTS

If a program calculation in an integer expression exceeds the range of an integer number, ZBasic will return the overflowed integer remainder of that calculation. The result will be incorrect. **ZBasic does not return an Integer Overflow Error.** Check program logic to insure results of an operation remain within integer range.

HOW INTEGER VARIABLES ARE STORED IN MEMORY

Integer variables and integer array elements require two bytes* of memory. To find the address (location in memory) of an integer variable:

ADDRESS¹ = VARPTR(INTEGER VARIABLE[(SUBSCRIPT[,SUBSCRIPT[,Ö.]])])

ADDRESS² = ADDRESS¹ + 1

The value of **INTEGER VARIABLE** is calculated using this equation:

INTEGER VARIABLE=VALUE OF ADDRESS²*256 + VALUE OF ADDRESS¹



*Requires four bytes for LongInteger. The MSB and LSB are stored in reverse order with regular integers. See the Macintosh appendix for more information.

VARIABLES

FLOATING POINT (BCD) VARIABLES

There are three floating point precisions that may be configured by the programmer to return accuracy up to 54 significant digits:

ZBasic does all BCD calculations in DOUBLE PRECISION. This is extremely important when speed is a factor. If you only need 6 or 7 digits of precision and speed is important be sure to CONFIGURE DIGITS OF ACCURACY AS FOLLOWS:

DOUBLE PRECISION = 6
SINGLE PRECISION = 4
SCIENTIFIC PRECISION = 4

This setting will give you maximum speed in BCD floating point. See the appendix for your computer for variations or enhancements. This is not a factor for the optional binary math package available for some version of ZBasic.



The Macintosh accuracy can be configured up to 240 digits. Optimum BCD speed is realized by configuring double precision to 8, single and scientific precision to 6.



Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

DEFINING VARIABLES AS SINGLE OR DOUBLE PRECISION

To force the precision of a specific variable to be single precision, follow every occurrence of that variable with an exclamation point (!).

To force a variable to be double precision, follow the variable name with a pound sign (#).

To force ZBasic to define a range of variables as double or single precision, use the DEFDBL or DEFSNG statement:

DEFDBL A-G Makes all variables beginning with A-G as Double precision.
 A# and A would be the same variable in this case.

DEFSNG C Makes all variables beginning with C as Single precision.
 C! and C would be the same variable.

Note: Some versions of BASIC default to single precision variables instead of integer. Use DEFSNG A-Z in programs being converted or configure to assume Floating Point. Also see "Optimize Expression Evaluation as Integer" under "Configure".

VARIABLES

SCIENTIFIC - EXPONENTIAL NOTATION

ZBasic expresses large numbers like:

50,000,000,000
like this: **5E+10** or **5E10**

The plus sign (+) after the "E" indicates the decimal point moves to the right of the number. Ten places in this example.

Technically: 5×10^{10} or 5×10^{10} .

ZBasic expresses very small numbers like:

.000005
like this: **5E-06**

A minus sign after the "E" indicates the decimal point is moved to the left of the number that many places, six in this example. Technically: $5/10^{10}$ or 5×10^{-6} .

<u>STANDARD NOTATION</u>	<u>SCIENTIFIC NOTATION</u>
9,123,000,000,000,000	9.123E+15 (or E15)
-3,400,002,000,000,000,000	-3.400002E18 (or E+18)
.000,000,000,000,000,000,011	1.1E-20
-.000,012	-1.2E-05

Note: Some BASICs use scientific notation with a "D" instead of an "E". (like 4.23D+12 instead of 4.23E+12) ZBasic will read old format values correctly but will use the more common "E" when printing scientific notation.

WHEN SCIENTIFIC NOTATION IS EXPRESSED

Constants and variables will be expressed in scientific notation when the value is less than .01 or exceeds 10 digits to the left of the decimal point.

You can force ZBasic to print all significant digits in regular notation with: **PRINT USING**

See **PRINT USING** in the Reference Section of this manual.

RANGE OF ZBASIC FLOATING POINT VARIABLES

The range of floating point numbers, regardless of the accuracy configured is:

+ -1E-64 to + -9.9E+63.*

The digits of accuracy are 14 digits for double and 6 digits for single (this is the default for most systems and may be set by the user).



Double Precision exponent may range from E-16,384 to E+16,383. Single Precision exponent is the same for compatibility with 8 and 16 bit machines.

VARIABLES

OVERFLOW RESULTS

If an expression results in a number greater than +9.999E+63, a result of 9.999E+63 will be returned.

If the number is less than +-1.0E-64 the result will be zero. **ZBasic will not give an overflow or underflow error.** Check program logic so that numbers do not exceed floating point range.

BCD FLOATING POINT SPEED

To obtain maximum speed out of BCD floating point math be sure to configure the digits of accuracy to:

DOUBLE PRECISION = 6
SINGLE PRECISION = 4
SCIENTIFIC PRECISION = 4

Normally these settings are fine at 14 and 6 digits. They should only be changed when speed is extremely important. Converting equations to integer will greatly increase speed as well. These settings are important because ZBasic does all calculations in Double precision. Single precision is used for saving memory only.



Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

SINGLE AND DOUBLE PRECISION DIGITS OF ACCURACY

The only difference between Single and Double Precision is that Single Precision holds fewer significant digits than Double Precision. **ALL ZBASIC FLOATING POINT CALCULATIONS ARE PERFORMED IN DOUBLE PRECISION.**

The default digits of accuracy are 6 digits for Single Precision and 14 digits for Double Precision. The accuracy is configurable from 6 to 54 digits for Double and 2 to 52 digits for Single Precision.*

<u>ACTUAL NUMBER</u>	<u>SINGLE PRECISION*</u>	<u>DOUBLE PRECISION*</u>
12,000,023	12000000	120000023
.009,235,897,4	9.2359E-03	9.2358974E-03
988,888	988,888	988,888
.235,023,912,323,436,129	.235024	.23502391232344
9,999,999 .999,900,001,51	10000000	9999999.9999
88.000,000,912,001,51	88	88.000000912002
12.34147	12.3415	12.34147



*Defaults are 8 and 12 digits for the Macintosh. Both are configurable up to 240 digits.

VARIABLES

ROUNDING

If the digit just to the right of the least significant digit is greater than 5, it will round up, adding one to the least significant digit.

In the example for **.009,235,898,4** above, the last significant 6 digit number is nine, but since the digit after 9 is 7, the 9 is rounded up by one to 10 (and subsequently the 8 is rounded up to 9 to give us 9.2359E-03, which more accurately represents the single precision value. See "Configure" for ways of setting the rounding factor.

NUMBER DEFAULT ROUNDING FACTOR IS: 49

####49	.49+.49 = .98 which is less than one	No Rounding
####50	.50+.49 = .99 which is less than one	No Rounding
####51	.51+.49 = 1 which is equal to one	Rounds up
####52	.52+.49 = 1.1 which is greater than one	Rounds up

This rounding option will not be available for optional binary floating point packages.

CONFIGURING ACCURACY

ZBasic allows the user to configure the digits of accuracy for single, double or scientific precision functions (like LOG, TAN, SIN, etc.)

LIMITATIONS:

Double precision must be at least 2 digits more significant than single. Digits of Accuracy must be in multiples of two (four with Macintosh).

<u>TYPE</u>	<u>MINIMUM DIGITS</u>	<u>MAXIMUM DIGITS</u>
<u>PRECISION</u>	<u>OF ACCURACY</u>	<u>OF ACCURACY*</u>
SINGLE	2 DIGITS	2 DIGITS less than Dbl.
DOUBLE	6 DIGITS	54 DIGITS
SCIENTIFIC	2 DIGITS	54 DIGITS



*Note: **All floating point calculations are done in DOUBLE PRECISION.** For programs where floating point speed is important be sure to set the digits of accuracy to:

DOUBLE PRECISION	= 6
SINGLE PRECISION	= 4
SCIENTIFIC PRECISION	= 4

Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

WARNING: Programs sharing disk files and CHAINED programs with single or double precision variables must have the same accuracy configuration. If one program is set for 6 and 14 digits, and another program is set for 10 and 20 digits, the programs will not be able to read and write each others files.



Configurable up to 240 digits. For hi-speed set Double to 8, single and scientific to 6.

VARIABLES

ACCURACY AND MEMORY REQUIREMENTS

The number of bytes of memory or disk space required for storing single and double precision variables is dependent on the digits of accuracy. If you do not change the accuracy, ZBasic will assume 6 digits for single precision (which requires 4 bytes), and 14 digits for double precision (which requires 8 bytes).*

When you change accuracy, disk files, variables, and constants memory requirements will change as well. The equation to calculate memory or disk file space required for single or double precision variables is:

Digits of Accuracy / 2+1=Bytes required per Floating Point variable

<u>DIGITS of ACCURACY</u>	<u>DISK FILE AND VARIABLE MEMORY REQUIREMENTS</u>
2 digits	2 bytes
4 digits	3 bytes
5 digits	Will round odd digits UP to the next even number, 6 here
6 digits	4 bytes (Single precision default if not configured by user)
.	
.	
.	
14 digits	8 bytes (Double precision default if not configured by user)
.	
.	
.	
52 digits	27 bytes
54 digits	28 bytes



*The Macintosh defaults to 8 digits for single (four bytes) and 12 digits for double (eight bytes). Digits of accuracy are configurable in multiples of four (instead of two as above). To figure memory: Digits of Accuracy/2+2=bytes required.



WARNING: Different ZBasic programs sharing files and CHAINED programs MUST be set to the same accuracy. Failure to do this will result in program errors, faulty data reads or program crashes.

Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

VARIABLES

HOW BCD FLOATING POINT VARIABLES ARE STORED IN MEMORY

Single precision default is 6 digits (4 bytes). Double precision default is 14 digits (8 bytes). To locate the address (memory location) of either a Single or Double precision variable:

ADDRESS¹=VARPTR(FLOATING POINT VARIABLE [(SUBSCRIPT[,SUBSCRIPT[,...]])])

Single and Double precision variables are stored in Binary Coded Decimal format (BCD).

	Bit	7	6	5	...	0
*ADDRESS ¹ =						
	Bit 7:	Mantissa sign (0=POSITIVE, 1=NEGATIVE)				
	Bit 6:	The exponent sign (0=E+, 1=E-)				
	Bit 5-0:	The exponent value (0 to 64)				
ADDRESS2		Digit 1 and 2 (Four bits for each digit)				
ADDRESS3		Digit 3 and 4				
ADDRESS4		Digit 5 and 6 (Single precision default)				
ADDRESS5		Digit 7 and 8				
ADDRESS6		Digit 9 and 10				
ADDRESS7		Digit 11 and 12				
ADDRESS8		Digit 13 and 14 (Double precision default)				
.						
.						
ADDRESS28		Digit 53 and 54 (Limit of significant digits)				



*Single precision defaults to 4 bytes (six digits) and Double precision defaults to 8 bytes (12 digits). Macintosh computers use two bytes for mantissa and exponent for its high precision double precision variable type:

ADDRESS1 & 2	Bit	15	14	13	...	0
	Bit 15:	Mantissa sign				
	Bit 14:	Exponent sign				
	Bit 13-0	Exponent value 0-16383				

Range of 32 bit double precision is +-1.0E-16,383 to +-9.999E+16,384

Note: Single precision range is the same on all machines.



Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

VARIABLES

ACCURACY VERSUS PROCESSING SPEED

While ZBasic is capable of configuration to extremely high accuracy, you should be aware that calculation time is in direct relation to the number of digits of accuracy.

The following chart will clarify the relationship of processing time to accuracy.

ACCURACY versus PERFORMANCE

Math Function	Relative Speed	Digits of Accuracy						
		4/6*	6/6*	14	24	36	54	INTEGER
Add/Subtract	1		2/3	1	1.20	1.50	2.0	1/77
Multiply	3		1/7	1	1.25	3.10	5.8	1/33
Divide	12		1/6	1	1.25	1.75	3.0	1/33
SQR	50	1/5	1/4	1	2.50	5.75	13.0	
SIN	70	1/5	1/4	1	2.50	5.75	13.0	See USR8(0)
COS	70	1/5	1/4	1	2.50	5.75	13.0	See USR9(0)
TAN	150	1/5	1/4	1	2.50	5.75	13.0	
EXP	100	1/5	1/4	1	2.50	5.75	13.0	
LOG	65	1/5	1/4	1	2.50	5.75	13.0	
ATN	80	1/5	1/4	1	2.50	5.75	13.0	
X^n	140	1/5	1/4	1	2.50	5.75	13.0	
X^(integer)	30		1/2	1	1.67	2.75	5.0	
Shift <<,>>	2		3/4	1	1.25	1.75	2.2	1/20

EXPLANATIONS OF HEADINGS

- Math Function** The type of math function being timed.
- Relative Speed** All speeds are relative to ADD and SUBTRACT (SQR takes 50 times longer than add and subtract). The numbers also correspond to the approximate time (in milliseconds) it takes to perform 14 digit math on a Z80 at 4 MHZ.
- Digits of accuracy** The numbers under the digits are all relative to 14 digit accuracy. Examples: 54 digit divide takes 3 times longer than 14 digit 6 digit divide takes 1/7th the time of 14 digit multiply.
- INTEGER** Integer calculations are relative to 14 digit processing time. Integer add and subtract operations take 1/77th the time of 14 digit operations.
- *4/6** Scientific Accuracy operations were set for LOG, TAN, EXP, ^, SIN, COS and ATN only. Other functions remain at double precision.

SPEED To obtain maximum speed with BCD floating point calculations, configure the digits of precision to : DOUBLE PRECISION=6, SINGLE PRECISION=4, SCIENTIFIC PRECISION=4. ZBasic does ALL calculations in DOUBLE PRECISION.



Important Note: Some versions of ZBasic offer an optional high speed binary-floating-point option. While the speed of binary math packages is superior, the accuracy, range and memory requirements of binary math are much different from the standard BCD math described above. See the manual provided with the binary math package for details.

VARIABLES

STRING VARIABLES

String variables are used for storing alphanumeric, symbol, and control characters.

ZBasic string variables may hold up to a maximum of 255 characters. Any character with an ASCII code in the range of zero to 255 may be used. ASC(A\$) will return zero if A\$ is a null string: IF LEN(A\$)>0 AND ASC(A\$)≠ 0 THEN ASCII CODE=0

STRING, NUMBER CONVERSIONS

VAL	Converts a string to a number: X=VAL(A\$)
STR\$	Converts a number to a string: A\$=STR\$(43)
CVI, CVB	Converts a condensed string to a number
MKI\$, MKB\$	Converts numbers to condensed strings.



See DEFSTR LONG for using CVI and MKI\$ with LongIntegers

DEFINING STRING VARIABLES

Use a \$ symbol following a variable name to make it a string variable. A\$ will always be a string variable because of the \$.

To define a range of variables beginning with a certain character to be string variables (so you do not have to use \$ every time), use the statement DEFSTR:

DEFSTR A-M	Makes all variables starting with A, B, C. up to M as string variables. A is the same as A\$
DEFSTR X,Y,Z	Makes all variables starting with X,Y and Z as string variables. Z is the same as Z\$.

STRING VARIABLE ASSIGNMENTS

String variables are assigned alphanumeric values like this:

A\$="Hello there"	
ART\$="VanGogh"+" DaVinci"	(+) connects the strings (concatenates)
Z\$=B\$	
Z\$=B\$+C\$	
Z\$="Hello"+C\$+TEST\$	
MID\$(A\$, 2, 3)="YES"	Puts "YES" into A\$ starting at position 2

VARIABLES

STRING FUNCTIONS AND RELATED COMMANDS

String variables are used for storing and manipulating character information. Here are some examples of ZBasic's string capabilities:

<u>STRING FUNCTIONS</u>	<u>DEFINITION</u>
DIM 10 A\$	sets the string variable A\$ to a length of ten.
DEF LEN 20	Sets the following strings to 20 character length.
W\$=LEFT\$(A\$,3)	W\$= 3 characters from the left of A\$.
W\$=RIGHT\$(A\$,1)	W\$= 1 character from the right of A\$.
B\$=MID\$(A\$,4,2)	B\$=2 characters from A\$ beginning at position 4.
MID\$(A\$,2,3)=B\$	Puts first 3 characters of B\$ into A\$ starting at position 2.
C\$=CHR\$(65)	C\$= the character represented by ASCII 65 (letter A).
X=ASC("A")	X= the ASCII code of "A" (65).
X=INSTR(2,A\$,B\$)	Looks for B\$ in A\$ starting at position 2, and makes X equal to the position if found, otherwise X=zero.
A\$=STR\$(2345)	Makes A\$ equal "2345"
X=VAL(A\$)	Makes X equal the VALue of A\$ (2345 if above).
X=LEN(A\$)	X= the number of characters in A\$.
INPUTA\$	Gets input from the keyboard and stores it in A\$.
LINEINPUTA\$	Accepts any keyboard characters, stores them in A\$ and terminates input only with the <ENTER> key.
A\$=INKEY\$	Makes A\$= the last key pressed without using <ENTER>
A\$=UCASE\$("Hello")	Converts A\$ to UPPERCASE. (A\$ now equals "HELLO").
X=VARPTR(A\$)	X= the memory address of the variable A\$.
WRITE#1,A\$,20	Writes 20 characters of A\$ out to the disk file#1.
READ#1,A\$,20	Reads 20 characters off the disk into A\$.
A\$=STRING\$(10,"#")	Makes A\$ equal to "#####".
PRINT SPACE\$(4)	PRINTs 4 spaces.
SWAP A\$,B\$	Make A\$ equal B\$ and B\$ equal A\$.
LPRINTA\$	Prints A\$ out to the printer.
PRINT A\$	Prints A\$ to the screen.
PRINT #2,A\$	Prints A\$ to disk file 2.
OPEN"R",1,F\$,129	Opens the random access file named F\$.
KILL A\$	Erases the file specified by A\$ off the storage device.
A\$=DATE\$	Puts the date into A\$ (MM/DD/YY) (Most systems).
A\$=TIME\$	Puts the time into A\$ (HH/MM/SS) (Most systems).
A\$=B\$+C\$	Makes A\$ equal to B\$ plus C\$ (Concatenates).
A\$="HI"+"THERE"	Makes A\$ equal to "HI THERE".
PSTR\$	Special command to avoid duplication of string constants.

SPECIAL INDEX\$ COMMANDS

INDEX\$(n)="simple string"	INDEX\$="Simple string"
INDEX\$I(n)=A\$	INSERT A\$ at INDEX\$(n), moves up all other elements.
INDEX\$D(n)	DELETE element (n) of INDEX\$ and move up other elements.
X=INDEXF(A\$)	Looks for A\$ in INDEX\$ (all) X equals element if A\$ found. Else equals -1.
X=INDEXF("END",950)	Look for "END" in INDEX\$ starting at the 950th element.
CLEAR nnnnn	Set aside nnnnn bytes for INDEX\$.
CLEAR INDEX\$	Nullify the contents of the entire INDEX\$ array.

VARIABLES

STRING CONDITIONALS

Strings may be compared using conditional operators just like numbers. The difference is that they are compared by the value of the ASCII code for that number. For instance, the ASCII code for "A" is 65 and "B" is 66. Therefore the expression "A"<"B" would be true (-1).

See ASCII Chart in your computer manual. ASCII characters may vary from computer to computer and from printer to printer.

Be aware that ZBasic differentiates between upper and lowercase characters. "a" is greater than "A" because the ASCII code for "a" is 97 and the ASCII code for "A" is 65. If you want ZBasic to look at a string variable as uppercase only, use the UCASE\$ function to convert it.

ZBasic "looks" at all the characters in a string when doing comparisons. "Aa" is greater than "AA". "AAAAAa" is greater than "AAAAAAA" etc. ZBasic will compare characters in a string to the last character in that string.

<u>CONDITION</u>	<u>RESULT</u>
"RRRRR"<"S"	True (-1)
"FRANK"="FRANK"	True (-1)
"abc">"ABC"	True (-1)
TEST\$="Hello" (If TEST\$="Hello")	True (-1)
"A">"B"	False (0)
"YES"="yes"	False (0)

SIMPLE STRINGS

Quoted string: "Hello", "This is within quotes"

String variable: A\$, NAME\$, FF\$, BF\$(2,3)

Any of the following string commands: MKI\$, MKB\$, CHR\$, HEX\$, OCT\$, BIN\$, UNS\$, STR\$, ERRMSG\$, TIME\$, DATE\$, INKEY\$, INDEX(n)

COMPLEX STRINGS

May be any combination of SIMPLE STRINGS.

String operations containing one of the following commands: simple- string + simplestring. LEFT\$, RIGHT\$, MID\$, STRING\$, SPACE\$, UCASE\$ would be a complex string.

COMPLEX STRINGS MAY NOT BE USED WITH IF-THEN STATEMENTS.

ZBasic allows only one COMPLEX STRING per statement. If you wish to perform more than one complex sting at a time, simply divide the complex string expression into multiple statements like this:

CHANGE complex strings

B\$=RIGHT\$(A\$+C\$,2)

B\$=UCASE\$(LEFT\$(A\$,3))

IF LEFT\$(B\$,2)="IT" THEN 99

TO simple strings

B\$=A\$+C\$: B\$=RIGHT\$(B\$,2)

B\$=LEFT\$(A\$,3): B\$=UCASE\$(B\$)

D\$=LEFT\$(B\$,2): IFD\$="IT" THEN 99

VARIABLES

USING STRING VARIABLES EFFICIENTLY

String variables will require 256 bytes of memory for each string used if the string lengths are not defined by the user. It is important to realize that extensive use of string variables or string array variables may require the user to define string lengths to avoid running out of memory.

Note: Some BASIC(s) have what is referred to as "Garbage collection". ZBasic's method of storing strings NEVER creates time wasting "Garbage Collection".

DEFINING THE LENGTH OF STRING VARIABLES

ZBasic strings have a default length of 255 characters. This can cause excessive memory usage. To obtain maximum memory efficiency, there are two ways of defining the length of string variables and string array variables:

DEF LEN = *number* (Numbers only. No expressions.)
DIM *number* **STRING VARIABLE**, or *number* **STRING ARRAY**, ...

DEFINING STRING LENGTHS WITH DIM

```
DIM X$(10), 20 A$, Z$(5), 45 TEST$, 10 MD$(20,20)
```

In this example the strings are allocated:

X\$(10)	255 each element (255 is the default. 2816 bytes)
A\$	20 (21 bytes)
Z\$(5)	each element of Z\$ as 20* (21*6=105 total bytes of memory used.)
TEST\$	45 (46 bytes)
MD\$(20, 20)	each element of MD\$(20,20) as 10. (21 * 21 *11=4851 total bytes of memory used.)

* If no length is defined, the last given length in that DIM statement is used (20 for A\$ in this example). If no length was defined in that DIM statement then the DEFined LENgth is assumed (255 if the string length has not been previously defined)

Note: Add one to the defined length of each string to determine the actual memory requirement of the string PLUS ONE for the LENGTH BYTE.

VARIABLES

DEFINING STRING LENGTHS WITH DEFLEN

Another command for DEF(ining) the LEN(gth) of string variables is:

```
DEF LEN = NUMBER (No expressions)
(In the range of 1 to 255)
```

Each string variable located AFTER the statement will have that length, unless another DEFLEN or DIM statement is used.

```
DIM A$(9,9), X(99), H$(999), 4Bull$
DEF LEN=50:B$="HOPE"
C$="HELLO"
DEF LEN=100
ART$="COOL"
DIM Coolness$(9)
A$=ART$
```

In the example:

A\$(9,9)	allocated 255 characters for each array element (ZBasic automatically allocates 255 if length has not been defined).
Bull\$	allocated 4 characters.
B\$ and C\$	allocated 50 characters each.
ART\$	allocated 100 characters
Coolness\$	allocated 100 characters for each element.
A\$	allocated 100 characters.

Note: The actual memory required for each string (each string element in an array) is the defined length plus one byte for the length byte.

VARIABLES

HOW STRING VARIABLES ARE STORED IN MEMORY

ADDRESS=VARPTR(STRING VARIABLE[(SUBSCRIPT[,SUBSCRIPT[,Ö.]])])

ADDRESS	Length Byte: Holds number of characters in the string.
ADDRESS+1	First character of the string variable
ADDRESS+2	Second character
.	
.	
.	
ADDRESS+n	Last character of the string variable
ADDRESS+255	Last address available for undefined string variable
ADDRESS+Defined Length	Last address available for defined string variable



WARNING 1: Strings should never be assigned a character length longer than the assigned length. If the length of A\$ is 5 and a program line is executed that has: A\$="1234567890", the characters "6" through "0" will overwrite the variables following A\$, possibly causing system errors or faulty data.

WARNING 2: If using INPUT to input strings with set length, always make sure the string length is at least one longer than the length being used for input.

For most versions of ZBasic, no error is generated if string assignments exceed the length of the string.



See "Configure" in the Macintosh appendix for setting string length error checking.

INDEX\$





SPECIAL INDEX\$ STRING ARRAY

INDEX\$ is a special ZBasic string array with some powerful and unique capabilities.

The following commands work with INDEX\$ variables only.

<u>INDEX\$ COMMAND</u>	<u>MEANING</u>
INDEX\$(n) =simple string	Assigns a value to INDEX\$(n)
INDEX\$ I(n) =simple string	Move element n and all consecutive elements up one and INSERT simple string at element n (the value in element 3 moves up to element 4). Actually inserts the value into the array without destroying any other elements.
INDEX\$ D(n)	DELETE element n and move all consecutive elements back down to fill the space (value in element 4 moves down to element 3).
X=INDEXF (simple string [,start#])	FIND simplestring in INDEX\$. Begin looking at element START#. If found X=element number If not found X = -1.

USING INDEX\$

INDEX\$ array variables may be assigned values like other string variables. To illustrate the power of INDEX\$, the following values have been stored into INDEX\$ elements INDEX\$(0) through INDEX\$(3) and will be used in the examples on the following pages:

<u>ELEMENT #</u>	<u>DATA</u>
INDEX\$(0)=	"AL"
INDEX\$(1)=	"BOB"
INDEX\$(2)=	"DON"
INDEX\$(3)=	"ED"

INDEX\$

INSERTING ELEMENTS INTO INDEX\$

INDEX\$ I (n) To INSERT "CHRIS" into INDEX\$, between "BOB" and "DON", you would use the command INDEX\$ I(2)="CHRIS".

This instructs ZBasic to move "DON" and "ED" down and insert "CHRIS" in element 2. (INDEX\$ I(2)=A\$ would also be legitimate) INDEX\$ would now look like this:

<u>ELEMENT #</u>	<u>DATA</u>
INDEX\$(0)=	"AL"
INDEX\$(1)=	"BOB"
INDEX\$(2)=	"CHRIS"
INDEX\$(3)=	"DON"
INDEX\$(4)=	"ED"

DELETING ELEMENTS FROM INDEX\$

INDEX\$ D (n) To DELETE "BOB" from INDEX\$ use the command INDEX\$ D(1). This instructs ZBasic to delete element one, and move "CHRIS" and "DON" and all the other elements up to fill in that space. The INDEX\$ array would now look like this:

<u>ELEMENT #</u>	<u>DATA</u>
INDEX\$(0)=	"AL"
INDEX\$(0)=	"CHRIS"
INDEX\$(0)=	"DON"
INDEX\$(0)=	"ED"

FIND A STRING IN INDEX\$

X=INDEXF(simplestring [,element n])

ZBasic will begin searching from element n (element zero if not specified) for the string specified by simple string. Examples:

IF FOUND

X=ELEMENT NUMBER

IF NOT FOUND

X=NEGATIVE ONE(-1)

To FIND "DON" in the above list let's say that A\$="DON". Using the command X=INDEXF(A\$), X would return 2 to show that "DON" is in element 2 of INDEX\$.

To FIND "CHR" (part of "CHRIS"), you would use the command X=INDEXF("CHR"). X would return with the value of 1 since a match was found in the first three characters of "CHRIS".

If you tried to FIND "RIS": X=INDEXF("RIS"), X would return with a value of -1 (negative one) since the FIND command begins the search at the first character of each element, which MUST be significant ("C" must be part of the search).

If the command had been INDEXF("CHRIS", 3), X would have equaled -1 since the search began at element 3 and "CHRIS" is at element 1 it would never find "CHRIS."

INDEX\$ MEMORY REQUIREMENTS

INDEX\$ variable elements use memory only if there are characters stored in that element and only as much memory as needed to hold those characters (plus one for length byte). CLEAR nnnnn is used to allocate memory for INDEX\$. CLEAR INDEX\$ will clear (nullify) the present contents of INDEX\$.

INDEX\$ LIMITATIONS

INDEX\$ may not be used with SWAP.

USES OF INDEX\$

INDEX\$ is a valuable tool for disk indices, in-memory data bases, creating word processors, holding lists of strings with varying lengths and much more.

INDEX\$ is especially useful anytime unknown string elements lengths are needed.

USING INDEX\$ FOR AN INSERTION SORT

A good example of the power of INDEX\$ is using it to create a perpetual sort. It allows you to add items to a list instantly and always have the list in order:

```
CLEAR 10000: TRONB
DO
  INPUT "Input String":A$: GOSUB "INSERTION SORT"
UNTIL A$="END" <--- Type END to end inserting
GOTO "PRINT LIST"
:
"INSERTION SORT"
REM N=Number of items
REM A$= New to string to insert
:
B=N: S=0
DO
  H=(B-S+1)>>1.
  LONG IF A$ <= INDEX$(B-H)
    B=B-H
  XELSE
    S=S+H
  END IF
UNTIL B=S
INDEX$ I(B)=A$
N=N+1
RETURN
:
"PRINT LIST"
FOR X=1 TO N
  PRINT INDEX$(X)
NEXT
END
```

INDEX\$

HOW INDEX\$ ARRAY VARIABLES ARE STORED IN MEMORY

The INDEX\$ array is stored in memory in one contiguous block. The distance between each element is the number of characters in the string plus one byte for the length byte of the string.



WARNING: It is suggested that strings in INDEX\$ not be manipulated with PEEK and POKE.

Note: CLEAR is used on some computers to allocate memory for INDEX\$. CLEAR INDEX\$ is used to nullify the contents of INDEX\$.



This version has the ability to use up to ten INDEX\$ arrays at the same time. See appendix for details. Also see MEM(-1) for determining memory remaining for INDEX\$.

ARRAY VARIABLES



ARRAY VARIABLES

An Array variable is a multi-celled variable followed by coordinates for specifying which cell is to be used. The following is an example of a one dimension string array with 101 elements.

<u>ARRAY ELEMENT</u>	<u>VALUE</u>
NAME\$(0)=	"ABE"
NAME\$(1)=	"ADAM"
NAME\$(2)=	"ALEX"
NAME\$(3)=	"AMOS"
.	
.	
.	
NAME\$(100)	"ZORRO"

Separate variables could be used for each value, like NAME1\$="ABE", NAME2\$="ADAM" but typing a hundred different variables would become very tiring.

Array variables are much easier to use when inputting, saving, loading, printing long lists, moving data around in a list, sorting lists of information, etc. This example shows how easy it is to print a complete list of the names in the array of variables.

```
FOR X= 0 TO 100
  PRINT NAMES$(X)
NEXT
```

Computers are very good at manipulating large amounts of data and using regular variables to do this is very impractical.

MULTI-DIMENSIONED ARRAYS

ZBasic will allow arrays of 1,2,3 or more dimensions, depending on the amount of memory available on your computer.

ARRAY VARIABLES

TWO DIMENSION ARRAY EXAMPLE

The following chart shows a two dimensional integer array; A(3,3). The number of elements are determined by the BASE OPTION that was configured when loading ZBasic. The default is Base 0:

A(3,3) BASE 0 dimensions are 4 elements down (0,1,2 and 3) and 4 elements across (0,1,2 and 3). Base zero utilizes all the elements including the italicized.

A(3,3) BASE 1 dimensions are 3 elements down (1,2,3) and 3 elements across (1,2,3) (not the italicized):

<i>A(0,0)</i>	<i>A(1,0)</i>	<i>A(2,0)</i>	<i>A(3,0)</i>
<i>A(0,1)</i>	<i>A(1,1)</i>	<i>A(2,1)</i>	<i>A(3,1)</i>
<i>A(0,2)</i>	<i>A(1,2)</i>	<i>A(2,2)</i>	<i>A(3,2)</i>
<i>A(0,3)</i>	<i>A(1,3)</i>	<i>A(2,3)</i>	<i>A(3,3)</i>

This array was DIM(med) A(3,3). A(1,3) represents the cell underlined above. Accessing a cell only requires giving the correct coordinate after the variable name.

Variables, constants or expressions may be used in specifying coordinates:

A(3,2), A(X,Y), A(2,X), A(X*2/3,2+Y).

BASE OPTION

Zero is considered an element unless you set the BASE OPTION to one when configuring ZBasic. See "Configure" for more information about setting the Base option. The default BASE is zero.

DEFINING THE DIMENSIONS OF AN ARRAY

All variable arrays MUST be **DIM**ensioned at the beginning of a program. When you RUN a program, memory is set aside for the array based on the number of elements you have **DIM**ensioned.

An example of DIM:

```
DIM A%(10,10,10), A#(5), A!(9,7), B$(10), 5Cool$(20)
```

Only numbers may be used within DIM statement parentheses. The following DIM expressions are illegal:

```
DIM A(X), A(2*X), A(FR).
```

ARRAY VARIABLES

HOW ARRAYS USE MEMORY

The following chart shows how to calculate the memory requirements of the arrays DIMensioned above with a BASE OPTION of zero (default value).

<u>ARRAY</u>	<u>Type</u>	<u>Bytes per Element</u>	<u>How to Calculate**</u>	<u>Memory Required</u>
A%(10,10,10)	INTEGER	2	$11*11*11*2$	2662 Bytes
A#(5)	DOUBLE PREC.	8	$6*8$	48 Bytes
A!(9,7)	SINGLE PREC.	4	$10*8*4$	320 Bytes
B\$(10)	STRING	256	$11*256$	2816 Bytes
Cool\$(20)	STRING	6	$21*6$	126

**Note: If you use a BASE OPTION of ONE, you will not need to add one to the dimension. For instance, in the first example the way to calculate the memory required would be: $10*10*10*2$. Also see DEF LEN and DIM under STRING VARIABLES for info about defining sting lengths.



Macintosh also has LongInteger arrays. Each element takes 4 bytes.

ARRAY BOUNDS CHECKING

During the initial stages of writing a program, it is a good idea to configure ZBasic to check array bounds in runtime. See "Configure" for more information.

OUT OF MEMORY ERROR FROM DIMMING

It is necessary to have an understanding of how arrays use memory. DIMensioning an array larger than available memory will cause ZBasic to give an OUT OF MEMORY error at Compile time or RUN time. When calculating large arrays be sure to check if memory is sufficient.

ARRAY VARIABLES

PRINTING ARRAYS

Arrays were designed to make manipulating large lists of data easy. The following routines print the values of ARRAY(50) and/or ARRAY(50,5) to the screen (Substitute LPRINT for PRINT or use ROUTE 128 to print to the printer). Use AUTO or make your own line numbers. It does not matter which numbers are used.

“One Dimension array PRINT routine”

```
DIM ARRAY(50)
FOR X=0 TO 50
  PRINT ARRAY(X)
NEXT
```

“Two Dimension array PRINT routine”

```
DIM ARRAY(50,5)
FOR X=0 TO 50
  FOR X2=0 TO 5
    PRINT ARRAY(X,X2),
  NEXT X2
  PRINT
NEXT X
```

MAKING AN ENTIRE ARRAY ONE VALUE

The following examples show how to make an entire array (ARRAY(50) or ARRAY(50,5)) equal to a certain value. This would be convenient if you wanted to zero out an array or have all the elements start the same values.

“One Dimension array ASSIGNMENT routine”

```
DIM ARRAY(50)
FOR X=0 TO 50
  ARRAY(X)=VALUE
NEXT
```

“Two Dimension array ASSIGNMENT routine”

```
DIM ARRAY(50,5)
FOR X=0 TO 50
  FOR X2=0 TO 5
    ARRAY(X,X2)=VALUE
  NEXT X2
NEXT X
```

ARRAY VARIABLES

USING ARRAYS FOR SORTING

Arrays are also very convenient for organizing large lists of data alphabetically or numerically, in ascending or descending order.

The first program below creates random data to sort. This program is for example purposes only and should not be included in your programs. **These programs are included on your master disk.**

Follow the GOSUB with the label of the sort routine you wish to use (either "QUICK SORT" or "SHELL SORT"). Any line numbers may be used. These sort routines may be copied and saved to disk (using SAVE* or +) as a subroutine to be loaded with APPEND. See APPEND.

SORT.BAS FILL ARRAY WITH RANDOM DATA FOR SORTING

```
DIM SA(500), ST(30,1): REM ST (30,1) FOR QUICK SORT ONLY.
NI=500: REM Change DIM 500 and NI if sort larger
FOR X=0 TO NI
  SA(X)=RND(1000): REM Stores random numbers for sorting
NEXT
PRINT "Start Time: "; TIME$
GOSUB "QUICK SORT": REM Or SHELL SORT
PRINT "Finish Time: "; TIME$
FOR =NI-10 TO NI
  PRINT SA (X): REM Print last to make sure SORT worked.
NEXT
END
```

SHELL.APP SHELL-METZNER SORT

```
"SHELL SORT" Y=NI
"Z1" Y=Y/2
IF Y=0 THEN RETURN: REM Sort complete
Z99=NI-Y
FOR K9=1 TO Z99
  I=K9
  "X2" E2=I+Y
  REM: In line below change <= to >= for descending order
  IF SA ( I ) <= SA ( E2 ) THEN "X3" ELSE SWAP SA ( I ), SA ( E2 )
  I=I-Y
  IF I>0 THEN "X2"
"X3" NEXT K9
GOTO "Z1"
END
```

Note: To sort string arrays instead of numeric arrays add a "\$" to the appropriate variables.

Also see "Perpetual Sort" using INDEX\$ in the previous chapter.

ARRAY VARIABLES

```
QUICK.APP      QUICK SORT

"QUICK SORT"
REM Improve Quicksort submitted by Johan Brouwer, Luxembourg.
REM Thanks for the submission, Johan.
SP=0:ST(0,0)=0:ST(0,1)=0
ST(0,1)=NI
DO
  L=ST(SP,0): R=ST(SP,1):SP=SP-1
  DO
    LI=L: R1=R: SA=SA((L+R)/2)
    DO
      WHILE SA(LI) < SA
        LI=LI+1
      WEND
      WHILE SA(RI)>SSA
        RI=RI-1
      WEND
      LONG IF LI<= RI
        SWAP SA(LI), SA(RI)
        LI=LI+1:RI=RI-1
      END IF
    UNTIL LI>RI
    LONG IF (R-LI) >(RI-L)
      LONG IF L<RI
        SP=SP+1:ST(SP,0)=L: ST(SP,1)=RI
      END IF
      L=LI
    XELSE
      LONG IF LI<R
        SP=SP+1:ST(SP,0)=LI:ST(SP,1)=R
      END IF
      R=R1
    ENDIF
  UNTIL R<=L
UNTIL SP=-1
RETURN: REM QUICK SORT FINISHED HERE
END
```

Note: To use the QUICK SORT or SHELL SORT with STRING variables, use DEFSTR with the appropriate variables on the first line of the program or put a "\$" after all variables that are strings.

Be sure to use DEFLen or DIM to define the length of the string variables. If each element needs 50 characters, then set the length of SA\$ to 50. The default is 256 bytes per element for string variables if you do not define the length.

HINTS ON TYPING IN THE PROGRAM: First of all, use line numbers of your own choosing. Indentation in this program is the way ZBasic shows the loops or repetitive parts of the program. You do not need to type in spaces (Make everything flush left). ZBasic will indent the listing automatically when you type LIST or LLIST.

Also see "Perpetual Sort" using INDEX\$ in the previous chapter.

ARRAY VARIABLES

ARRAY ELEMENT STORAGE

The following chart illustrates how array elements for each type of variable are stored in memory.

Assumptions:

1. Memory starts at address zero (0)
2. Strings were dimmed: DIM 15 VAR\$(1,2,2) (Each element uses 16 bytes*)
3. Other arrays dimmed: DIM VAR%(1,2,2) VAR!(1,2,2), VAR#(1,2,2)
(SINGLE and DOUBLE precision assumed as 6 and 14 digit accuracy.)
4. BASE OPTION of ZERO is assumed.

Array ELEMENTS	RELATIVE ADDRESSES		SINGLE	DOUBLE
	STRING\$	INTEGER%	Precision!	Precision#
VAR(0,0,0)	00000	00000	00000	00000
VAR(0,0,1)	00016	00002	00004	00008
VAR(0,0,2)	00032	00004	00008	00016
VAR(0,1,0)	00048	00006	00012	00024
VAR(0,1,1)	00064	00008	00016	00032
VAR(0,1,2)	00080	00010	00020	00040
VAR(0,2,0)	00096	00012	00024	00048
VAR(0,2,1)	00112	00014	00028	00056
VAR(0,2,2)	00128	00016	00032	00064
VAR(1,0,0)	00144	00018	00036	00072
VAR(1,0,1)	00160	00020	00040	00080
VAR(1,0,2)	00176	00022	00044	00088
VAR(1,1,0)	00192	00024	00048	00096
VAR(1,1,1)	00208	00026	00052	00104
VAR(1,1,2)	00224	00028	00056	00112
VAR(1,2,0)	00240	00030	00060	00120
VAR(1,2,1)	00256	00032	00064	00128
VAR(1,2,2)	00272	00034	00070	00136

*Length byte adds one extra byte in front of each string element.

Note: Arrays are limited to 32,768 (0-32,767) elements.



LongInteger arrays are also supported. Each element takes four bytes. Macintosh is limited to 2,147,483,647 elements.



MSDOS version 4.0 has a limit of 32,768 (0-32,767) elements for integer arrays and a limit of 65,536 (0-65535) for string and floating point arrays.

GRAPHICS





GRAPHICS

Graphics are an extremely important way of communicating ideas. The old adage "A picture is worth a thousand words" is very true. ZBasic offers many powerful screen imaging commands and functions to take advantage of your computer's graphics capabilities.

In addition to having powerful graphic commands,, ZBasic defaults to utilizing the same graphic coordinates regardless of the system you happen to be programming on. This is ideal for moving programs from one machine to another without having to make changes to the graphic commands or syntax. Quite a change from the old days.

Definitions of some commonly used graphic terms:

- PIXEL** The smallest graphic point possible for a given system. Some systems allow you to set the color of a pixel.
- RESOLUTION** Refers to the number of pixels (dots of light) on a screen. A computer with a resolution of 400 x 400 has 160,000 pixels (high resolution). A computer with 40 x 40 resolutions has only 1600 pixels (low resolution).
- COORDINATE** By giving a horizontal and vertical coordinate you can describe a specific screen location easily. With ZBasic the origin (0,0) is the upper left hand corner of the screen or window.
- With a standard device independent coordinate system you can specify a location on the screen without worrying about pixel positions.

GRAPHICS

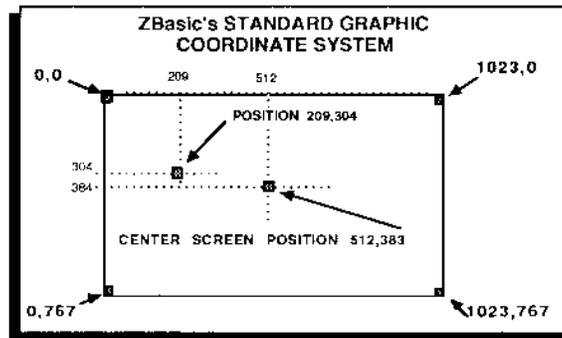
ZBasic's DEVICE INDEPENDENT GRAPHIC COORDINATE SYSTEM

ZBasic uses a unique DEVICE INDEPENDENT COORDINATE SYSTEM to describe the relative positions on a video screen, instead of a pixel system which describes specific graphic dots on the screen.



The standard coordinate system is 1024 points across (0-1023) by 768 points down (0-767). The width is broader to be in proportion to a normal video monitor.

This approach allows writing graphic programs the same way regardless of a computer's graphic capabilities.



Device independent graphics means the coordinate syntax is the same regardless of the device or type of graphics being used!

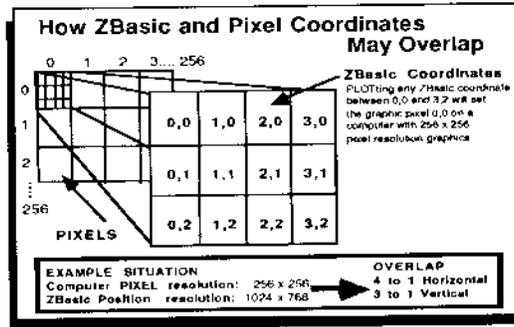
The ZBasic approach to graphics makes commands function the same way ***EVEN ON DIFFERENT COMPUTERS!*** ZBasic handles all the transformations needed to match up the ZBasic coordinates to the actual resolution of the computer. This is an ideal way of handling graphics in a standardized way.



On the Macintosh the standard coordinates apply to the current window, not to the screen. Macintosh and MSDOS versions of ZBasic have the extra commands; `COORDINATE` and `COORDINATE WINDOW` which allow you to set relative coordinates of your own or pixel coordinates, respectively. See the Apple appendix for ways of configuring ZBasic to pixel coordinates. Some Z80 See appendix for specifics.

SCREEN PIXEL versus SCREEN POSITION

It is important to realize that ZBasic's standard coordinate system of 1024 x 768 has a direct relation to the screen, NOT to the actual pixel resolution of the computer being used. It is important not to confuse the pixel coordinate with the position coordinate:

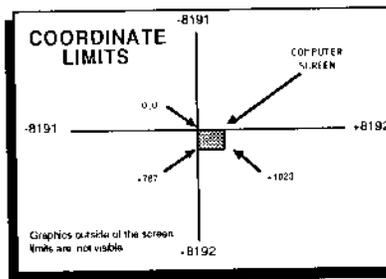


You can see that plotting coordinates; 0,0 through 3,2, sets the same pixel on a screen with 256 x 256 resolution. If the pixel resolution of a computer is 64 x 64 then PLOTTing 0,0 or 15,11 will plot the same pixel (16 to 1 horizontal and 12 to 1 vertical).

Fortunately this Information is rarely Important. ZBasic takes care of the tedious transformations between different graphic modes and resolutions. Skills learned on one machine may be used on any other machine that uses ZBasic!

OFF SCREEN COORDINATES

ZBasic allows coordinates to be given with graphic commands that are out of bounds of the actual screen coordinates. This allows drawing lines, circles or rectangles off the screen, with only that part of the graphics that are within bounds to be shown on the screen. ZBasic 'clips' the rest of the drawing.



The limits are from -8191 to +8192. Any coordinates given out of this range will cause an overflow and the actual result will be the overflowed amount without generating an error.

GRAPHICS

DIFFERENT TYPES OF GRAPHICS

Graphic appearance and quality will depend on the resolution of the computer or terminal you are using. Resolution is the number of graphics pixels on a screen. A computer with a resolution of 40 x 40 has 1600 different pixels. This is low resolution graphics because the graphic pints (pixels) are very large.

For computers without graphics, ZBasic will simulate the graphics as closely as possible using an asterisk. The resolution would be the number of characters across by characters down. See MODE.

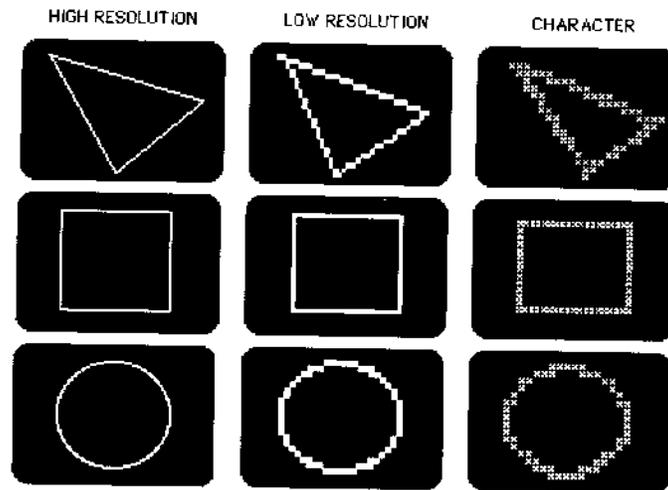
GRAPHICS TYPE

HIGH RESOLUTION
LOW RESOLUTION
CHARACTER

RESOLUTION

about 200 x 150 or More
about 150 x 100 or Less
TEXT graphics simulation.

A COMPARISON OF LOW AND HIGH RESOLUTION IMAGES

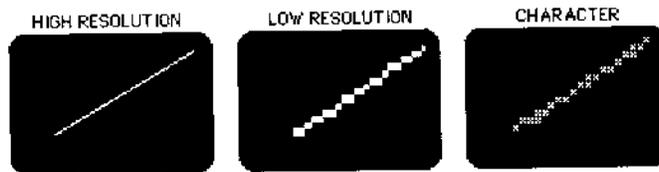


Notice the variation in quality. Programmers porting programs over to other machines should keep the resolution of the target computer in mind when creating programs.

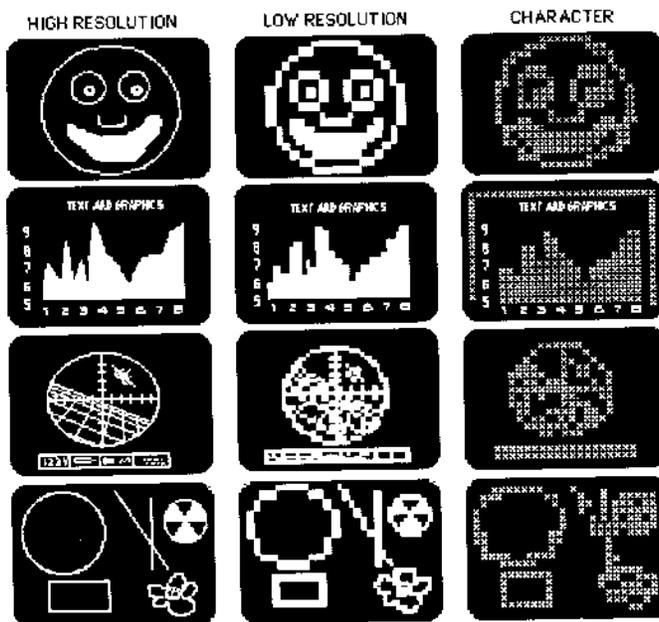
MORE GRAPHIC EXAMPLES AT DIFFERENT RESOLUTIONS

Quality deteriorates as graphic complexity increases and screen resolution decreases, although usually the lower the resolution the faster the execution speed. In this line example you can see the variation of quality.

The ZBasic statement to create all the lines in the first example was the same:
`PLOT 60,660 TO 1000, 10:`



Additional examples of more complex graphics forms in different resolutions:



GRAPHICS

MODE

ZBasic offers different modes of text and graphics output depending on hardware and model. The ability to change modes allows you to simulate the output for different machines. Syntax:

MODE expression

The following chart gives the modes for some popular microcomputers, and illustrates how modes are grouped according to resolution.

MODE CHART

Mode number	MSDOS type		APPLE //e, //c		TRS-80 I, III	
	Text	Graphic	Text	Graphic	Text	Graphic
0	40x25	character	40 x 24	character	32x16	character
1	40x25	40x40	none	40x48	64x16	128x48
2	80x25	character	80x24	character	32x16	character
3	80x25	80x25	none	80x48	64x16	128x48
4	80x25	character	40x24	character	32x16	character
5	40x25	320x200	40x24	280x192	64x16	128x48
6	80x25	character	80x24	character	32x16	character
7	80x25	640x200	80x24	560x192	64x16	128x48
8	40x25	character	40x24	character	32x16	640x240?
9	40x25	40x40	Bottom	40x48	64x16	128x48
10	80x25	character	80x24	character	32x16	character
11	80x25	80x25	Bottom	80x48	64x16	128x48
12	80x25	character	80x24	character	32x16	character
13	40x25	320x200	Bottom	280x165	64x16	128x48
14	80x25	character	80x24	character	32x16	character
15	80x25	640x200	Bottom	560x165	64x16	128x48

MACINTOSH	
Text	Graphic
Many Font styles and sizes here!	SEE Macintosh APPENDIX

CP/M-80	
Text	Graphic
Normally 80x24	SEE Z80 APPENDIX

Be sure to read the appropriate appendix for exact mode designations.



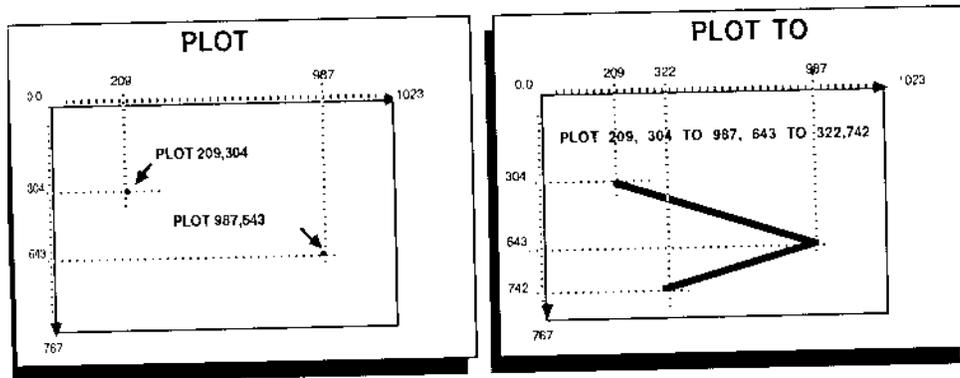
Note: Check your computer appendix for variations.

PLOTTING POINTS AND LINES

To set a specific screen position(s) to the current color or to draw lines from one screen position To another, TO another..., or to draw from the last screen position used (in another ZBasic statement) TO another...

PLOT [TO] horizontal , vertical [TO [horizontal , vertical [TO...]]]]

PLOT draws with the last color defined by COLOR. COLOR=0 is the background color of most computers, while COLOR=-1 is the foreground color. If you have a system with a black background, COLOR -1 is white and COLOR 0 is black. See COLOR in this chapter.



As with all other graphic commands, PLOT uses the standard ZBasic coordinates of 1024 x 768 regardless of the computer being used. When TO is used, ZBasic will plot a line from the first position TO the next position, TO the next position...

EXAMPLES OF PLOTTING

PLOT 4,5

RESULT

Turns on the pixel at the graphic position 4 positions over and 5 positions down.

PLOT 0,0 TO 1023,767

Plots a line from upper left corner of the screen down to the lower right corner of the screen.

PLOT TO 12,40

Draws a line from the last position used with the PLOT command TO the point on the screen 12 positions over by 40 positions down.

PLOT 0,0 TO 400,0 TO 0,300 TO 0,0

Plots a triangle in the upper left corner of the screen.

Note: All the examples above will plot in the current COLOR.

GRAPHICS

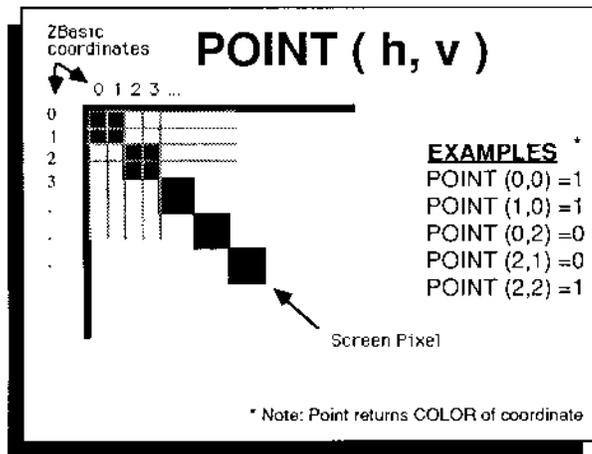
POINT

POINT (horizontal coordinate, vertical coordinate)

Returns the COLOR of the pixel at the ZBasic coordinate. Point is available on many computers to inquire about the COLOR of a specific screen graphic position (some computers do not have the capability to "see" pixels).

As with other commands, ZBasic Device Independent Graphic coordinates may overlap pixels. The following illustration shows the pixels and color types associated with them.

In this example: 0=BACKGROUND (WHITE) 1=FOREGROUND (BLACK)



As with all other ZBasic graphic commands the standard device independent coordinate system of 1024 x 768 is used.

Note: The ZBasic device independent coordinate system specifies positions on the screen, not pixels. See below for ways of setting your system to actual pixel coordinates, if needed.



Macintosh and **MSDOS** systems can be set to use pixel coordinates with COORDINATE WINDOW. See **Apple** appendix for ways of configuring to pixel coordinates. **Z80** see your hardware technical manual and the Z80 appendix for specifics of your machine.

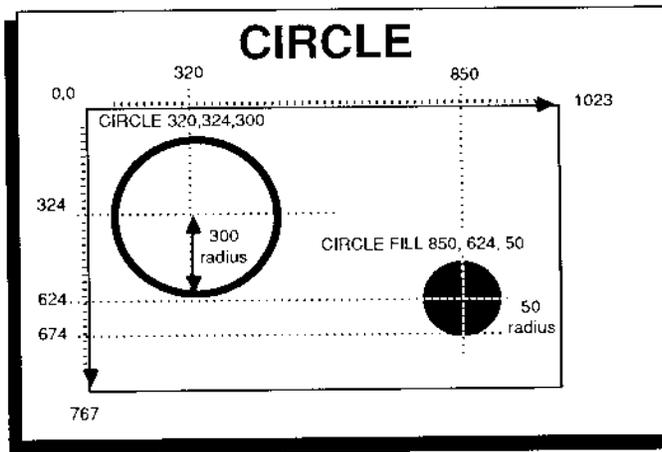
CIRCLE

CIRCLE [FILL] horizontal, vertical, radius

CIRCLE draws a circle in the currently defined COLOR and RATIO. COLOR=0 is the background color of most computers, while COLOR=-1 is the foreground color. If you have a system with a black background, COLOR -1 is white and COLOR 0 is black.

See RATIO for ways of changing the shapes of circles. Also see CIRCLE TO and CIRCLE PLOT for creating PIES and ARCS.

If FILL is used, the circle will be a solid ball in the current color.



As with all ZBasic graphic commands, the Device Independent Graphic Coordinates of 1024 x 768 are the default.



FILL is taken from PEN pattern; PEN,,,n. Where n is one of the pen patterns used under the control panel. Quickdraw circles are also available using toolbox calls. See appendix.

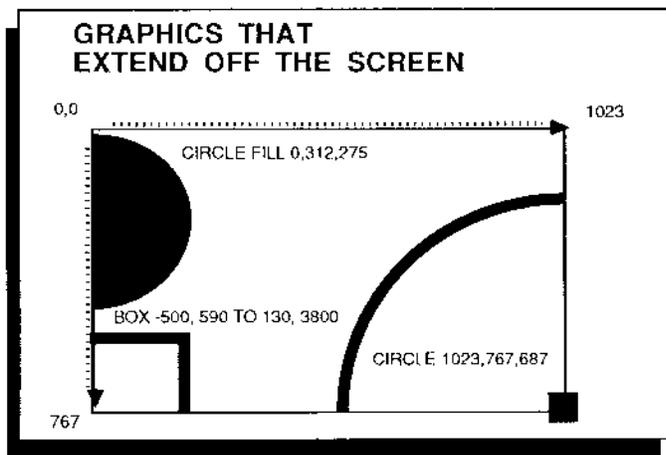
GRAPHICS

GRAPHICS THAT EXTEND OFF THE SCREEN (CLIPPING)

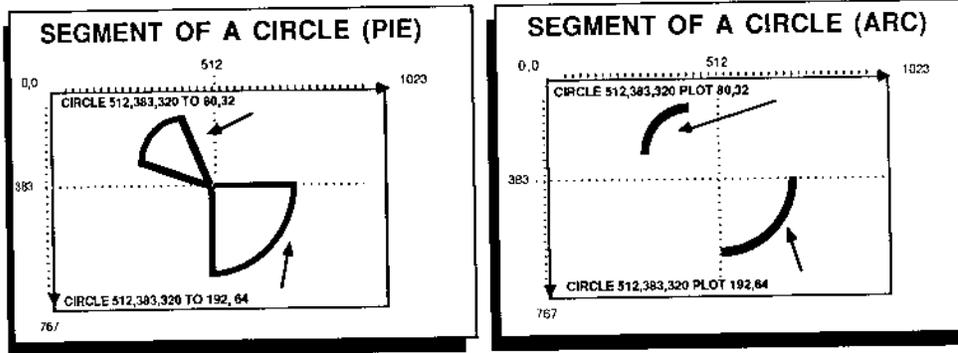
If coordinates are given that exceed the limits of the ZBasic screen coordinates, that part of the image exceeding the limits will be "CLIPPED".

It is still permissible to use these numbers and in many cases it is important to have them available for special effects.

CIRCLE, or other graphic commands like PLOT, BOX, PRINT% etc., with coordinates that are off the screen but are within the limits of -8192 to +8192 are permissible and that part out of range will be "clipped":



As with all ZBasic graphic commands, the Device Independent Coordinates of 1024 x 768 are used.



SEGMENT OF A CIRCLE (PIE)

To draw an enclosed segment of the circumference of a circle (PIE), use this syntax:

CIRCLE *h,v,radius* **TO** *starting BRAD degree, number of BRADs* (counter clockwise)

CIRCLE draws with the last color defined by COLOR. COLOR=0 is the background color of most computers, while COLOR=-1 is the foreground color. If you have a system with a black background, COLOR -1 is white and COLOR 0 is black. See COLOR in this chapter.

SEGMENT OF A CIRCLE (ARC)

To draw a segment of the circumference of a circle (an ARC) use the syntax:

CIRCLE *h, v, radius* **PLOT** *starting BRAD degree, number of BRADs* (counter clockwise)

CIRCLE draws with the last color defined by COLOR. COLOR=0 is the background color of most computers, while COLOR=-1 is the foreground color. If you have a system with a black background, COLOR -1 is white and COLOR 0 is black. See COLOR in this chapter.

Note: 256 BRADS=360 DEGREES. See the BRAD chart on the next page. As with all ZBasic graphic commands, the standard coordinates of 1024 x 768 are used.

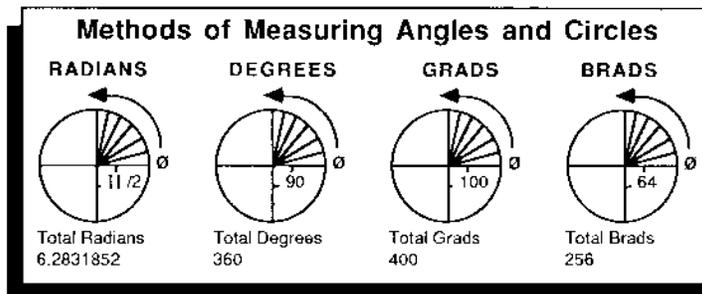
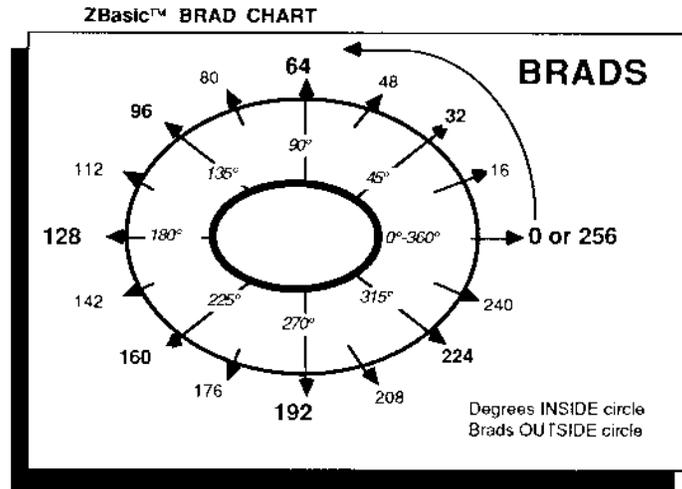


FILL may be used with the CIRCLE FILL *x,y,r, TO s,n* statement on this version. The FILL pattern is taken from PEN pattern; PEN,,,n. Where n is one of the pen patterns used under the control panel. Quickdraw arcs are also available using toolbox calls.

GRAPHICS

BRADS

Brads are used with ZBasic CIRCLE commands to determine a position on the circumference of a circle. Instead of DEGREES of zero to 359, BRADS range from zero to 255. (Starting at 3 O'clock going counter-clockwise.)



CONVERSIONS FROM ONE TYPE TO ANOTHER

RADIANS=DEGREES*ATN(1)/45
RADIANS=9*GRADS/10
RADIANS=BRADS/40.7436666

GRADS=10 * DEGREES/9
GRADS=RADIANS*63.66197723
GRADS=BRADS*1.5625

DEGREES=RADIANS*45/ATN(1)
DEGREES=BRADS*1.40625
DEGREES=GRAD/63.66197723

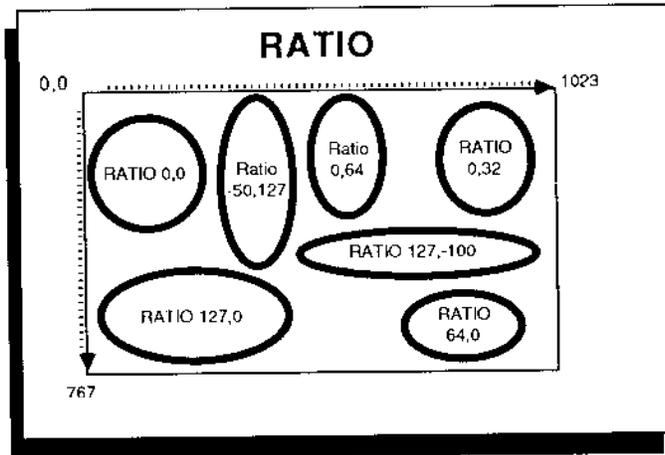
BRADS=DEGREES/1.40625
BRADS=GRADS/1.5625
BRADS=RADIANS*40.743666

Also see **USR8** and **USR9** for high-speed Integer SIN and COS.

RATIO

ZBasic allows you change the aspect ratio of any CIRCLE, ARC or PIE with the graphic statement RATIO:

RATIO *Width* (-128 thru + 127), *Height*(-128 thru +127)
(See CIRCLE)



Examples:

Ratio settings are executed immediately and all CIRCLE commands will be adjusted to the last ratio.

+127	=	2	times normal
+64	=	1.5	times normal
+32	=	1.25	times normal
0	=	0	Normal proportion
-32	=	.75	times normal
-64	=	.5	times normal
-96	=	.25	times normal
-128	=	0	(no width or height)



Quickdraw circles use box coordinates to set circle shape. See toolbox section of appendix.

GRAPHICS

BOX

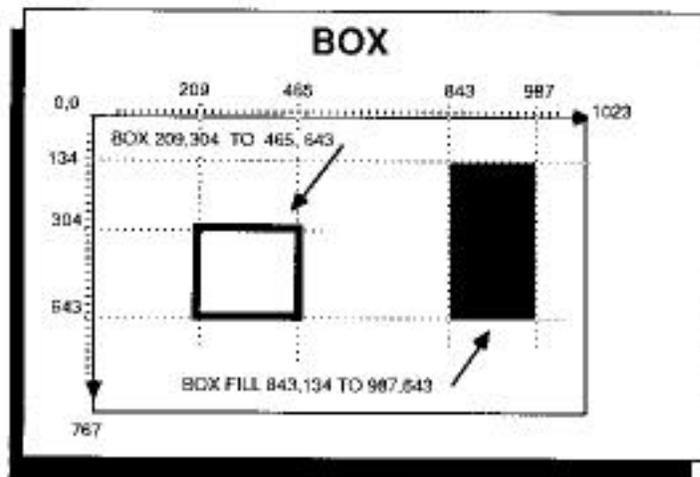
Box is used for drawing rectangles in the current color. The size of a rectangle is specified by giving the coordinates of opposing corners.

BOX [FILL] h1, v1 TO h2, v2

h1, v2 The first corner coordinate of the BOX.

h2, v2 The opposite corner coordinate of the BOX.

The BOX is plotted in the current color. If FILL is used the BOX will be filled with the current COLOR.



As with all ZBasic graphic commands, the device independent coordinates of 1024 x 768 are used. Notice the different quality of BOXes on various computers and different modes.



FILL is taken from PEN pattern; PEN,,,n. Where n is one of the pen patterns used under the control panel. Quickdraw boxes are also available using toolbox calls. See appendix.

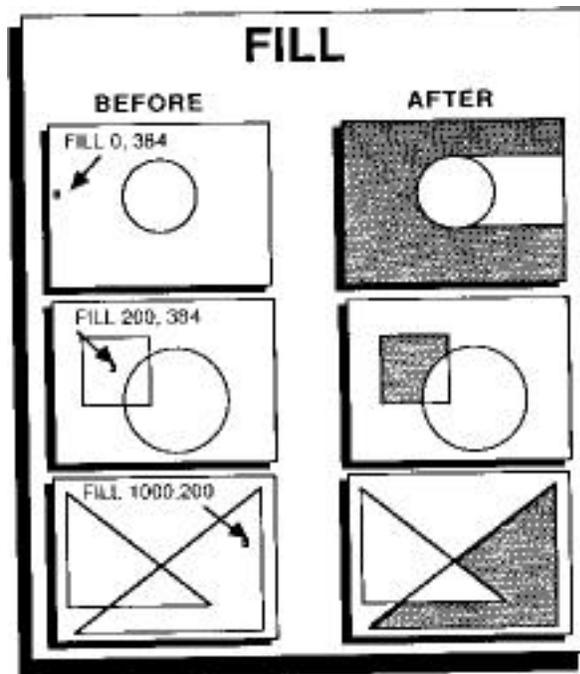
FILL

FILL Horizontal expression, Vertical expression

The fill command will fill a screen position from upper left most position it can reach without finding a color other than the background color, and down to the right and to the left until a non-background color is found.

This command will not function on computers lacking the capability to read screen pixel coordinates. See computer appendix.

Example:



As with all ZBasic graphic commands, the Device Independent Coordinates of 1024 x 768 are used.

Also see CIRCLE FILL and BOX FILL



FILL pattern is taken from PEN pattern; PEN,,,n. Where n is one of the pen patterns used under the control panel. A much faster way to fill screen segments is using Quickdraw FILL with polygons, circles and rectangles. See appendix.

GRAPHICS

COLOR

COLOR is used to signify the color to be used with PLOT, CIRCLE, BOX and FILL. All systems support zero and -1 for background and foreground colors. (BLACK and WHITE respectively on most systems).

COLOR [=] expression

The following chart represents the color codes for IBM PC and compatible systems with color graphics. Colors codes vary significantly from system to system so check your computer appendix for variations.

IBM PC and Compatible COLOR codes

0= BLACK	8= GRAY
1= BLUE	9= LIGHT BLUE
2= GREEN	10= LIGHT GREEN
3= CYAN	11=LIGHT CYAN
4= RED	12= LIGHT RED
5= MAGENTA	13= LIGHT MAGENTA
6= BROWN	14= YELLOW
7= WHITE	15= BRIGHT WHITE

Color intensities will vary depending on the graphics hardware and monitor being used. Check your computer appendix for variations.



While most Macintoshes are black and white, COLOR is useful when printing to the ImageWriter II with a color ribbon. See appendix for details.

CLS, CLSLINE, CLSPAGE

CLS is used to clear the entire screen of graphics or text quickly. Optionally, the text screen may be filled with a specific ASCII character (in most modes). Check your computer appendix for variations.

CLS [ASCII code:0-255]

CLS LINE is used to clear a text line of text and graphics from the current cursor position to the end of that line.

CLS LINE

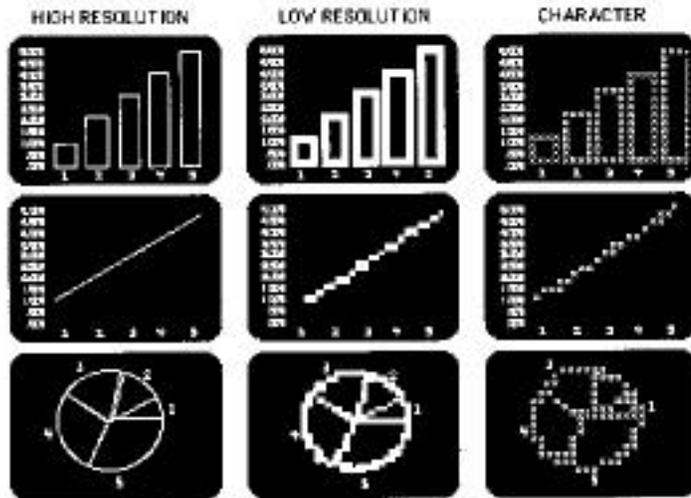
CLS PAGE is used to clear a text screen of text and graphics from the current cursor position to the end of the screen.

CLS PAGE

See Computer Appendix

BUSINESS GRAPHS, CHARTS ETC.

Business graphs and charts are easily accomplished with ZBasic graphics. An added benefit is that the graphs are also easily transported to different computers.



To further assist you in porting graph programs, ZBasic has two text commands that correspond to the graphic position on the screen instead of the text position:

PRINT%(h,v)

Prints from the position specified by the ZBasic graphic coordinates.

INPUT%(h,v)

Positions the input to be from the graphic position specified by h,v.

The syntax of these commands is the same as PRINT and INPUT. Also see PRINT@.

GRAPHICS

SPECIALIZED GRAPHICS

The Apple, MSDOS, Macintosh and some Z80 versions of ZBasic have some added powerful features for graphics. See the appendix for your version of ZBasic for specific information:

APPLE // GRAPHICS



Double Hi-Res with 16 colors is supported for the Apple //e, //c and //GS with 128k or more. Text and graphic may be integrated on the screen and customizable character sets are also supported. LONG FN's for DRAW, BLOAD and BSAVE are on the master disk.

IBM PC, MSDOS GRAPHICS



Version 4.0 supports most of the graphic modes of IBM PC's and compatibles including; Hercules Monochrome Graphics, Hercules PLUS, Enhanced Graphics Adaptor (EGA), Color Graphics Adaptor (CGA), Monochrome and all other graphics modes.

Also supported are GET and PUT graphic commands, PLOT USING, COORDINATE and COORDINATE WINDOW. See appendix for specifics.

MACINTOSH GRAPHICS



The master disk contains examples of printing and displaying MacPaint graphics and TIFF bit images. Also supported is GET and PUT graphics. PICTURE, TEXT, Apple's QuickDraw and toolbox routines, PEN and many more. See appendix for specifics.

TRS-80, CP/M-80 GRAPHICS



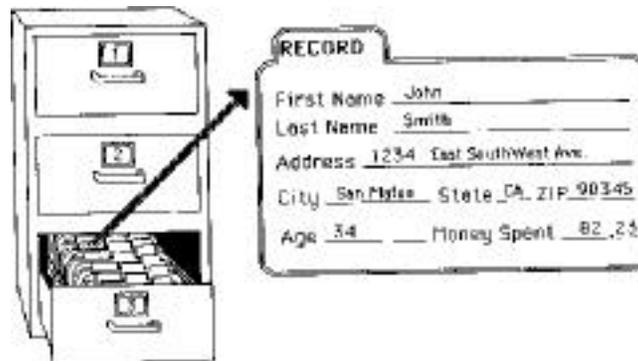
Most TRS-80 graphics are supported including Radio Shack's Hi-Res and Micro-Lab's Hi-Res boards on the Model 4 in MODE 8 and 15 (text and graphic integration is not supported with the Radio Shack Hi-Res board). Hi-Res is not supported on the model one or three.

Because of the diversity of machines for CP/M systems and because of a lack of common interface, graphics are not supported with CP/M systems (although we have special graphics versions for Kaypro 4 and 10 with graphics capabilities).



FILE HANDLING

ZBasic file commands are the same on all versions. This section explains file commands and statements. ZBasic file concepts are similar to a file cabinet:



EVERYDAY TERMS

FILE CABINET

Holds files in drawers.

FILE

Contains data for a mail list or inventory control system among other things.

RECORD

One logical part of a file: All the data for Mr. Smith in a mail list (name, address...)

PARTS OF A RECORD

One part of a Record: The address or the City in a mail list record.

ZBASIC TERMS

DISK OPERATING SYSTEM

Holds files on diskettes, cartridges etc.

FILENAME, FILENUMBER

Contains data for a mail list or inventory control system among other things.

RECORD

One logical part of a file: All the data for Mr. Smith in a mail list file (name, address...)

LOCATION

One part of a RECORD: The address in a mail list record or even one character in the address.

FILES

GLOSSARY OF ZBASIC FILE TERMS

DOS: The Disk Operating System is a program residing in a computer's memory which takes care of the actual reading, writing and file control on a storage device such as floppy drives, hard drives, tape backup devices, etc. ZBasic works with the formats and syntax of each disk operating system using its syntax for such things as filenames, drive specs, etc.

FILENAME: Tells ZBasic which file to access. A string constant or variable is used.

FILESPEC: The part of a filename (or some other indicator) that specifies the device, directory or sub-directory a file is on. See your DOS manual for correct filespec syntax.

FILENUMBER: ZBasic may be configured to have from 0 to 99 files OPEN at the same time (if DOS and available memory permit). Filenumbers are used in a program with disk file commands to instruct ZBasic which file is being referred to. For example; if you open a file called "Fred" as number one, when doing file commands you need only refer to file number one, not "Fred". This saves a lot of typing.

RECORD: A record is one segment of a file. A mail list record might include Name, Address, City, State, ZIP, etc. If you want data from a specific record, it is called up using the RECORD command. The first record in a ZBasic file is RECORD 0. There may be up to 65,535 RECORDs in a file.* RECORD #filename, record, location.

SEQUENTIAL METHOD: This is a method of reading a file one element or record at a time, in order ---one after another i.e. 1,2,3,.. .

RANDOM METHOD: This is the method of reading file items randomly--- out of order. i.e. RECORD 20,90,1,22

FILE POINTER: It is often important to know how to manipulate the file pointer. ZBasic allows you to position the file pointer by using RECORD, and tells you where the file pointer is currently positioned by using REC(filename) and LOC(filename).

COMPATIBILITY WITH MSBASIC™

Experienced BASIC programmers will like the power and simplicity of ZBasic file commands. For the first time, BASIC file handling commands have been made compatible and portable. All ZBasic disk commands function the same way regardless of the computer being used.

Sequential file commands are very similar. The main difference being that items written with PRINT# should be separated with quoted commas in ZBasic if being read back with INPUT#.

Random file commands have been made simpler, yet just as powerful. Those experienced with MSBASIC file commands should find the conversion painless:

ZBASIC COMMANDS

READ, WRITE, RECORD

PRINT#, INPUT#, LINEINPUT#

MSBASIC EQUIVALENTS

FIELD, GET, PUT, MKI\$, CVI, MKS\$,

CVS, MKD\$,CVD, LSET, RSET

PRINT#, INPUT#, LINEINPUT#

FILE COMMANDS COVERED IN THIS SECTION

This outline gives an overall perspective of file commands available in this section and groups commands in logical order. This section of the manual provides lots of examples and a tutorial for the file commands of ZBasic.

OPENING AND CLOSING FILES

- OPEN
- CLOSE

DELETING OR ERASING FILES

- KILL

RENAMING A FILE

- RENAME

POSITIONING THE FILE POINTER

- RECORD

WRITING TO A FILE

- WRITE#
- PRINT#
- PRINT#, USING
- ROUTE

READING FROM A FILE

- READ#
- INPUT#
- LINEINPUT#

GETTING IMPORTANT FILE INFORMATION

- LOF
- LOC
- REC



Be sure to read the appendix for your computer. Many versions have extra commands that take advantage of a particular system.

FILES

CREATING AND OPENING FILES

OPEN ["O, I or R"], filenumber, "filename" [,record length]

All ZBasic files must be opened before processing.

OPEN "O"

Opens a file for "O"utput only. If the file does not exist, it is created. If it does exist, all data and pointers are erased and it is opened as a new file.

OPEN "I"

Opens a file for "I"nput only. If the file does not exist, a "File Not Found" error is generated for that file number.

OPEN "R"

Opens a "R"andom access file for reading and/or writing. If the file does not exist, it is created. If the file exists, it is opened, as is, for reading or writing.

filenumber

ZBasic may be configured to have from 1 to 99 files open at one time in a program (depending on the DOS and available memory for that computer). Files are assigned numbers so ZBasic knows to which file it is being referred. The original copy of ZBasic is configured to allow up to two open files at a time. If you wish to have more files open, you may configure ZBasic for up to 99 open files. See "Configure".

record length

Record length is optional. If it is omitted, a record length of 256 characters is assumed. Maximum record length is 65,535 characters, or bytes (check appendix for variations).

EXAMPLES OF OPENING FILES

OPEN "O", 2, "NAMES", 99

Opens filenumber 2 as "NAMES", with a record length of 99 characters, for OUTPUT only. If "NAMES" doesn't exist, a file named "NAMES" is created. If a file called "NAMES" exists, all data and pointers in it are deleted and it is opened as a new file.

OPEN "I", 1, A\$

Opens filenumber 1 whose filename is the contents of A\$, with assumed record length of 256 for INPUT only. If A\$ doesn't exist, a "File Not Found" error is generated for filenumber one. See "Disk Error Trapping" for more information.

OPEN "R", 2, "BIGFILE", 90

Opens filenumber 2 named "BIGFILE", with a record length of 90, for Reading and Writing.



OPEN"IR", "OR", "RR" for resource forks. OPEN "A" for append also supported. Volume number is used after record number i.e. OPEN"R",1,"Fred",99, vol%. A number of other enhancements are covered in the appendix.

CLOSING FILES

CLOSE[# filename [, filename,...]]

All files should be closed when processing is finished or before ending a program. *Failure to close files may result in lost data.*

CLOSE without a filename closes all open files (STOP and END will also CLOSE all files). It is very important to close all opened files before exiting a program. When a file is closed, the end-of-file-marker is updated and any data in the disk buffer is then written to the disk.

After you close a file, that filename may be used again with another OPEN.

DELETING FILES

KILL "filename"

Files may be deleted from the disk from within a program or from the editor with the "KILL" command. From the editor the filename must be in quotes on Macintosh and Z80 versions.

Filename is a simplestring and may be represented by a string constant or variable:

```
TRONB
INPUT"FILE TO KILL: ";FILE$
INPUT"ARE YOU SURE? ";A$
IF A$<>"YES" THEN END
KILL FILE$
END
```

RENAMING FILES

RENAME "oldfilename" TO [or comma] "newfilename"

Files may be renamed on the disk from within a program or directly using RENAME.

Filenames may be a string constant or variable. Example:

```
TRONB
INPUT"FILE TO RENAME";OLDFILE$
INPUT"NEW NAME: ";NEWFILE$
RENAME OLDFILE$ TO NEWFILE$
```



The TRS-80 Model 1,3 version does not support RENAME.



Macintosh: Both KILL and RENAME also use Volume number. See appendix for syntax.
MSDOS: CHDIR and Pathnames may be used. APPLE ProDOS: Pathnames may be used.

FILES

WRITING TO A FILE USING PRINT#, WRITE# AND ROUTE#

PRINT#

PRINT# filename, (variables, constants or equations);", "...]

PRINT# is used for writing data in TEXT format. It is saved to the disk quite like an image is saved to paper using LPRINT. PRINT# is useful for many things but it is not the fastest way or most efficient way to save data. See WRITE# below. Examples:

PRINT#1, A\$;","; C\$;","; Z% ;","; X#

Prints A\$, C\$, Z%, and X#, to filename one starting at the current file pointer. A carriage return* is written after the X#. This command stores data the same way it would be printed. Syntax is compatible with older versions of BASIC. The file pointer will point at the location in the file directly following the carriage return.*

PRINT#1, USING "##.##"; 12.1

Formats output to filename one starting at the current file pointer (stores 12.10). Functions like PRINT USING.

*Data MUST be separated by a delimiter of a quoted comma or a carriage return if reading data back using INPUT#. Some systems write a carriage return and a linefeed (two bytes).

WRITE#

WRITE[#] filename, variable [, variable...]

WRITE# is used for storing data in condensed format at the fastest speed. WRITE# may only be used with variables and data is read back with the READ# statement. Example:

WRITE#1, A\$;10, Z%, K\$;2

Writes 10 characters from A\$, the value of Z%, and 2 characters from K\$ to filename one, starting at the current file pointer. In the example; A\$;10 stores A\$ plus enough spaces, if any, to make up ten characters (or truncates to ten characters if longer).

ROUTE#

ROUTE [#] device

ROUTE is used to route output to a specific device. Device numbers are:

0 video monitor (default) **1-99** DISK filename (1-99)
128 PRINTER (same as LPRINT) **-1 or -2** SERIAL port 1 or 2*

Example of routing screen data to a disk file or serial port:

1. Open a file for output (use OPEN "C" and -1 or -2 for serial ports)
2. ROUTE to filename or serial port number that was opened.
 all screen PRINT statements will be routed to the device specified.
3. ROUTE 0 (so output goes back to the video)
4. Close the file or port using: CLOSE# n.

* Be sure to see your computer appendix for specifics.

READING FROM A FILE USING INPUT#, LINEINPUT# AND READ#

INPUT#

INPUT# filename, variable[, variable...]

INPUT# is used to read text data from files normally created with PRINT#. The data must be read back in the same format as it was sent with PRINT#. When using PRINT# be sure to separate data items with quoted comma or carriage return delimiters, otherwise data may be read incorrectly or out of sequence. Example:

INPUT#1, A\$, C\$, Z%, X#

Inputs values from filename one from the current RECORD and LOCATION pointer, into A\$, C\$, Z%, and X#. In this example the data is input which was created using the PRINT# example on the previous page. The file pointer will be pointing to the next location after X#.

LINEINPUT#

LINEINPUT# filename, variable (**One variable only**)

LINEINPUT# is used primarily for reading text files without the code limitations of INPUT#. Commas, quotes and other many other ASCII characters are read without breaking up the line. It will accept all ASCII codes accept carriage returns or linefeeds. TEXT is read until a carriage return or linefeed is encountered or 255 characters, whichever comes first:

LINEINPUT#5, A\$

Inputs a line into A\$ from filename five from the current file pointer. Accepts all ASCII codes including commas and quotes, except linefeed (chr10) and carriage return (chr 13). Terminates input after a chr 13, chr 10, End-of-file, or 255 characters.

READ#

READ [#] filename, variable [, variable...]

READ# is the counterpart of WRITE#. It is used to read back data created with WRITE# in condensed high-speed format. This is the most efficient way of reading files. Example:

READ#1, A\$;10, Z%, K\$;2

Reads 10 characters into A\$, an integer number into Z%, and 2 characters into K\$ from filename one, from the current file pointer. The file pointer will be pointing to the location directly following the last character in K\$ (includes trailing spaces if string was less than ten).

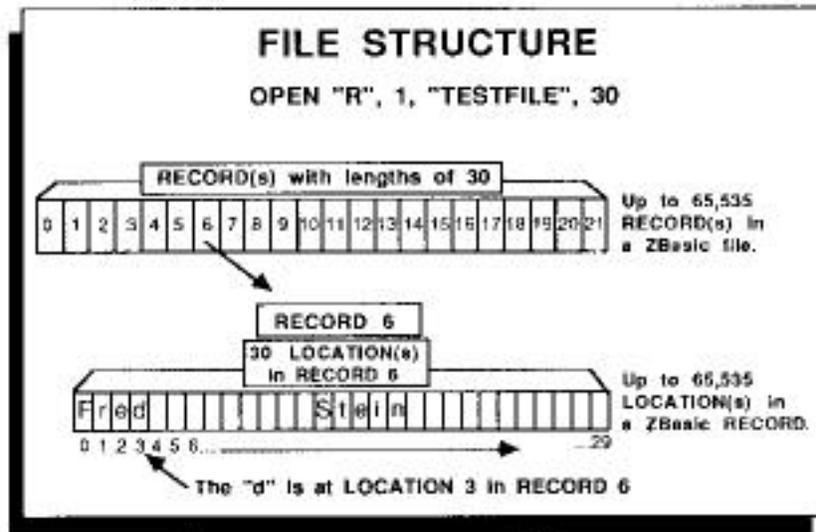
GETTING IMPORTANT INFORMATION ABOUT A SPECIFIC FILE

<u>Syntax</u>	<u>Description</u>
REC(filename)	Returns the current RECORD number location for filename.
LOC(filename)	Returns the current location within the current RECORD for filename (the byte offset).
LOF(filename)	Returns the last RECORD number of filename. If there are one or zero records in the file, LOF will return one. Due to the limitations of some disk operating systems this function is not always exact on some systems. Check the COMPUTER APPENDIX for specifics.

FILES

ZBASIC FILE STRUCTURE

All ZBasic files are a contiguous string of characters and/or numbers (bytes). The order and type of characters or numbers depends on the program that created the file.



In the illustration, the name "Fred Stein" was stored in RECORD six of "TESTFILE". To point to the "d" in FILENUMBER 1, RECORD 6, LOCATION 3 use the syntax:

```
RECORD#1, 6, 3
```

The location within a record is optional, zero is assumed if no location is given. If RECORD 1, 6 had been used (without the 3), the pointer would have been positioned at the "F" in "Fred" which is LOCATION zero.

If RECORD is not used, reading or writing starts from the current pointer position. If a file has just been OPEN(ed), the pointer is at the beginning of the file. (RECORD#n,0,0)

After each read or write, the file pointer is moved to the next available position in the file.



Macintosh: RECORD length and number of records is 2,147,483,647.

POSITIONING THE FILE POINTER

RECORD [#] filename, RECORD number [, LOCATION number]

To point to any LOCATION in any RECORD in any FILE, use:

RECORD 3,23,3 Sets the pointer of filename 3 to RECORD 23, LOCATION 3. If RECORD 23 contained "JOHN", then LOCATION 3 of this record would be "N", since zero is significant.

RECORD #3,23 Sets the pointer for file#3 to location zero in RECORD 23. If RECORD 23 contained JOHN, the character being pointed at would be "J".

RECORD IS OPTIONAL

If the RECORD statement is not used in a program, the pointer will have a starting position of RECORD 0, LOCATION 0 and is automatically incremented to the next position (for reading or writing) depending on the length of the data.

FILE SIZE LIMITATIONS*

The file size limitations for sequential files are either the physical limitations of the storage device or the limit of the Disk Operating system for that computer.

The limitation for Random access files is 65,536 records with each record containing up to 65,536 characters. Maximum file length is 4,294,967,296 characters (although multiple files may be linked to create larger files).

It is important to note that most Disk Operating Systems do not have this capability. Check your DOS manual for maximum file sizes and limitations.



Macintosh: RECORD length and number of records is 2,147,483,647.

CONFIGURING THE NUMBER OF FILES IN A ZBASIC PROGRAM

If the number of files is not configured, ZBasic assumes only 2 files will be used and sets aside only enough memory for two files.

To use more than 2 files, configure ZBasic for the number of files you need under "Configure".

ZBasic allows the user to configure up to 99 disk files for use in a program at one time (memory and disk operating system permitting). Each type of computer requires a different amount of buffer (memory) space for each file used so check your computer appendix for specifics (usually there are 256-1024 bytes allocated per file; 10 files would require between 2,560-10,240 bytes).

*See computer appendix for variations.

SEQUENTIAL METHOD



SEQUENTIAL METHOD



SEQUENTIAL METHOD

This section covers some of the methods that may be used when reading or writing files sequentially. It covers the use of READ, WRITE, PRINT#, INPUT# and LINEINPUT#.

SEQUENTIAL METHOD USING PRINT# AND INPUT#

These two programs demonstrate how to create, write, and read a file with PRINT# and INPUT# using the Sequential Method:

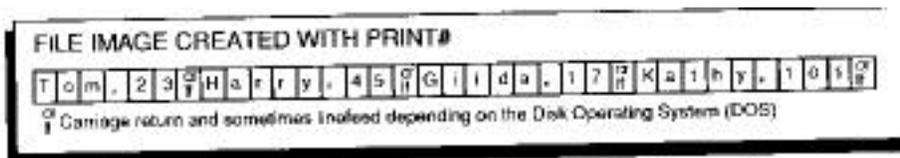
PRINT#

```
OPEN"O",1,"NAMES"  
DO: INPUT"Name: ";NAME$  
   INPUT "Age: "; AGE  
   PRINT#1, NAME$, "AGE"  
UNTIL NAME$="END"  
CLOSE#1: END
```

INPUT#

```
OPEN"I",1,"NAMES"  
DO: INPUT#1, NAME$, AGE  
   PRINT NAME$, "AGE"  
UNTIL NAME$="END"  
CLOSE#1: END
```

Type "END" to finish inputting names in the PRINT# program. The INPUT# program will INPUT the names until "END" is read.



Unless a semi-colon is used after the last data being printed to the disk, the end of each PRINT# statement is marked with a carriage return.

PRINT# USING

USING is used to format the PRINT# data. See "PRINT USING".

COMMAS IN PRINT# AND INPUT#

It is important to remember when using PRINT# with more than one data item, that quoted commas (",") must be used to set delimiters for data being written. If commas are not quoted, they will merely put spaces to the disk (as to the printer) and INPUT# will not be able to discern the breaking points for the data.

SEQUENTIAL METHOD

SEQUENTIAL METHOD USING READ# AND WRITE#

Other commands which may be used to read and write sequential data are READ# and WRITE#. The main difference between READ#--WRITE# and PRINT#--INPUT# is that the latter stores numeric data and string data, much the same way as it appears on a printer; READ# and WRITE# store string and numeric data in a more condensed and predictable format. In most cases this method is also much faster.

VARIABLES MUST BE USED WITH READ# AND WRITE#

READ# and WRITE# require that variables be used for data. Constants or expressions may not be used with these commands except the string length, which may be an expression, constant or variable.

HOW STRINGS ARE STORED USING WRITE#

When using WRITE# or READ# with strings, you must follow the string variable with the length of the string:

```
WRITE#1,A$;10,B$;LB      READ#1, A$;10, B$;LB
```

An expression may be used to specify the string length and must be included. When WRITE#ing strings that are shorter than the specified length, ZBasic will add spaces to the string to make it equal to that length. If the string is longer than the length specified, it will be "Chopped off" (If the length of A\$ is 20 and you WRITE#1,A\$;10, the last 10 characters of A\$ will not be written to the file).

Normally, you will READ# strings back exactly the same way you WRITE# them. Notice that the spaces become a part of the string when they are READ# back. If you WRITE# A\$;5, and A\$="HI" when you READ# A\$;5, back, A\$ will equal "HI " (three spaces at the end of it). The length of A\$ will be 5.

To delete the spaces from the end of a string (A\$ in this example), use this statement directly following a READ# statement:

```
WHILE ASC(RIGHT$(A$,1))=32: A$LEFT$(A$,LEN(A$)-1): WEND
```

You can use READ# and WRITE# using variable length strings as well. See the two format examples on the following pages.

SEQUENTIAL METHOD

READ# AND WRITE# IN CONDENSED NUMBER FORMAT

Numbers are stored in condensed format when using READ# and WRITE#. This is done to conserve disk space AND to make numeric space requirements more predictable. ZBasic automatically reads and writes condensed numbers in this format. Just be sure to read the data in exactly the same order and precision with which it was written. Space requirements by numeric variable type are as follows:

<u>PRECISION</u>	<u>MAXIMUM DIGITS</u>	<u>SPACE REQUIRED</u>
INTEGER	4.3 (+-32,767)	2 bytes
SINGLE PRECISION	6 (default)	4 bytes
DOUBLE PRECISION	14 (default)	8 bytes

Since single and double precision may be configured by the user, use this equation to calculate the disk space required if different than above:

$$(\text{Digits of precision} / 2) + 1 = \text{number of bytes per variable}$$



LongInteger has 9.2 digits and requires 4 bytes for storage. To calculate the storage needs for Macintosh Double precision; $\text{Digits}/2+2=\text{space required per variable}$.

INTEGER NUMBER CONVERSIONS

For those programmers that want to control conversions these commands are available. They are not required with READ and WRITE since these commands do it automatically.

X=CVI (simplestring) Converts the first two bytes of simple-string to integer (X).

A\$=MKI\$ (integer) Converts an integer to a 2 byte string

SINGLE AND DOUBLE PRECISION NUMBER CONVERSIONS

For those programmers that want to control conversions these commands are available. They are not required with READ and WRITE since these commands do it automatically.

X#=CVB (simplestring) Converts up to the first 8 bytes* of simplestring to an uncondensed double precision equivalent and stores the value in X#. (If string length is less than eight characters, only that many characters will be converted. At least two bytes are needed.)

A\$=MKB\$ (X#) Converts a Double precision number to an 8 byte string.*

X!=CVB (simplestring) Converts the first 4 bytes* of simplestring into a single precision number and stores the value in X! If string length is less than eight characters, only that many characters will be converted. At least two bytes are needed.

A\$=MKB\$ (X!) Converts a single precision number to a 4 byte string.*

*Note: The number of bytes of string space in the conversions depends on the precision set by the user. Use the equation above for calculating the space requirements. ZBasic assumes 8 bytes for double precision and 4 bytes for single precision if the user does not set precision.



To manipulate LongIntegers with MKI\$/CVI use DEFSTR LONG. See Macintosh appendix.

SEQUENTIAL METHOD

SEQUENTIAL FILE METHOD USING READ# AND WRITE#

The following programs illustrate how to use READ# and WRITE# using the sequential file method.

USING READ# AND WRITE# WITH SET LENGTH STRINGS

The programs below create and read back a file with the sequential method using READ# and WRITE#. String length is set to 10 characters by the "10" following NAME\$. ZBasic adds spaces to a string to make it 10 characters in length, then saves it to the disk.

AGE is assumed to be an integer number since it was not defined and is stored in condensed integer format.

WRITE#

```
OPEN"O",1,"NAMES"  
DO: INPUT"Name: "; NAME$  
   INPUT"Age: "; AGE  
   WRITE#1,NAME$;10, AGE  
UNTIL NAME$="END"  
CLOSE#1: END
```

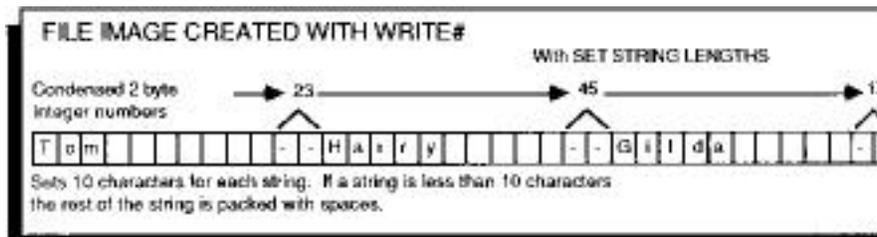
READ#

```
OPEN"I",1,"NAMES"  
DO: READ#1, NAME$;10, AGE  
   PRINT NAME$;" ";AGE  
   A$=LEFT$(NAME$,3)  
UNTIL NAME$="END"  
CLOSE#1:END
```

Type "END" to finish inputting names for the WRITE# program. The READ# program will READ the names until "END" is encountered.

FIXED STRING LENGTH WRITE#

This illustration shows how strings saved with set lengths appear in a disk file:



The reason the ages 23, 45 and 17 are not shown in the file boxes is because the numbers are stored in condensed format (2 byte integer).

SEQUENTIAL METHOD

USING READ# AND WRITE# with VARIABLE LENGTH STRINGS

READ# and WRITE# offer some benefits over PRINT# and INPUT# in that they will read and write strings with ANY ASCII characters. This includes quotes, commas, carriage returns or any ASCII characters with a code in the range of 0-255. The following programs will save strings in condensed format, using the amount of storage required for each string variable.

WRITE#

```
OPEN"O",1,"NAMES"  
DO: INPUT"Name: "; NAME$  
   INPUT"Age: "; AGE  
   LB$CHR$(LEN(NAME$))  
   WRITE#1, LB$;1  
   WRITE#1, NAME$;ASC(LB$),AGE  
UNTIL NAME$="END"  
LAST$="END":  
WRITE#1, LAST$; 3; CLOSE#1  
END
```

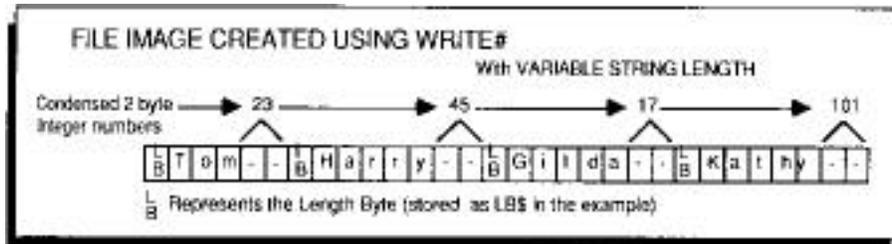
READ#

```
OPEN"I",1,"NAMES"  
REM  
DO: READ#1, LB$;1  
   READ#1, NAME$;ASC(LB$), AGE  
   PRINT NAME$, "AGE"  
UNTIL A$="END"  
CLOSE#1  
END
```

The WRITE# program stores a one byte string called LB\$ (for Length Byte). The ASCII of LB\$ (a number from 0 to 255) tells us the length of NAME\$.

VARIABLE STRING LENGTH WRITE#

This illustration shows how the data is saved to the disk when string data is saved using the variable length method. LB for "Tom" would be 3, LB for "Harry" would be 5, etc...



SEQUENTIAL METHOD

APPENDING DATA TO AN EXISTING FILE CREATED USING THE SEQUENTIAL METHOD

Sometimes it is faster (and easier) to append data to the end of an existing text file, instead of reading the file back in, and then out again.

This may be accomplished by using "R", for random access file when opening the file, and keeping track of the last position in a file using REC(filename) and LOC(filename) and putting a character 26 at the end of the file.

To append sequentially to a text file created with other programs try using this example program. The key is setting the record length to the right amount. The MS-DOS version uses 128. Other versions will vary.

This example creates a function called: FN Open(f\$, F%) and will OPEN the file named f\$, with file number f%, for appending. The RECORD pointer will be positioned to the next available space in the file.

To close a file properly for future appending, use the function called FN Close (f\$,f%).

```
LONG FN Open (f$,f%): REM FN OPEN(f$, f%)
  OPEN "R", f%, f$,128:REM Change 128 to correct # for your DOS
  FileLen%=LOF(f%): NextRec%=FileLen%: NextLoc%=0
  LONG IF FileLen%>0
    NextRec%=NextRec%-1
    RECORD #f%, NextRec%, NextLoc%
    READ #f%, NextRec$;128: REM Change this 128 too!
    NextLoc%=INSTR(1,NextRec$,CHR$(26)): REM (zero on Apple)
    IF NextLoc%>0 THEN NextLoc%=NextLoc%-1 ELSE NextRec%=NextRec%+1
  END IF
  RECORD #f%, NextRec%, NextLoc%
END FN
```

```
LONG FN Close (f$, F%)
REM TCLOSE the file correctly with an appended chr 26.
PRINT#f%, CHR(26);
CLOSE#f%
END FN
```

Note: This method will work with ASCII text files ONLY!



See Open "A" in the appendix for opening files for Append.



CREATING FILES USING THE RANDOM ACCESS METHOD

Random access methods are very important in disk file handling. Any data in a file may be stored or retrieved without regard to the other data in the file. A character or field from the last record in a file may be read (or written) without having to read any other records.

A simple example of the Random access method is the following program that reads or writes single characters to any LOCATION in a file:

RANDOM ACCESS EXAMPLE USING A ONE BYTE RECORD LENGTH

```
OPEN "R" , 1 , "DATA" , 1
REM RECORD LENGTH = 1 character

"Get record number"
DO: INPUT "Record number: ";RN
  INPUT "<R>ead, <W>rite, <E>nd: ";A$
  IF A$="R" GOSUB "Read" ELSE IF A$ = "W" GOSUB "Write"
UNTIL A$="E": CLOSE#1: END

"Write"
INPUT "Enter character: " ` A$
RECORD #1, RN
WRITE #1,A$;1 :RETURN

"Read"
RECORD #1,RN :REM Point at record# RN
READ #1,A$;1
PRINT" Character in RECORD# "; RN ;" was " ;A$: RETURN
```

To change this program to one that would read or write people's names, merely change the RECORD LENGTH to a larger number and increase the number after the A\$ in the READ# and WRITE# statements.

The following pages will demonstrate a more practical use of the Random Access method by creating a mail list program in easy to understand, step by step procedures.

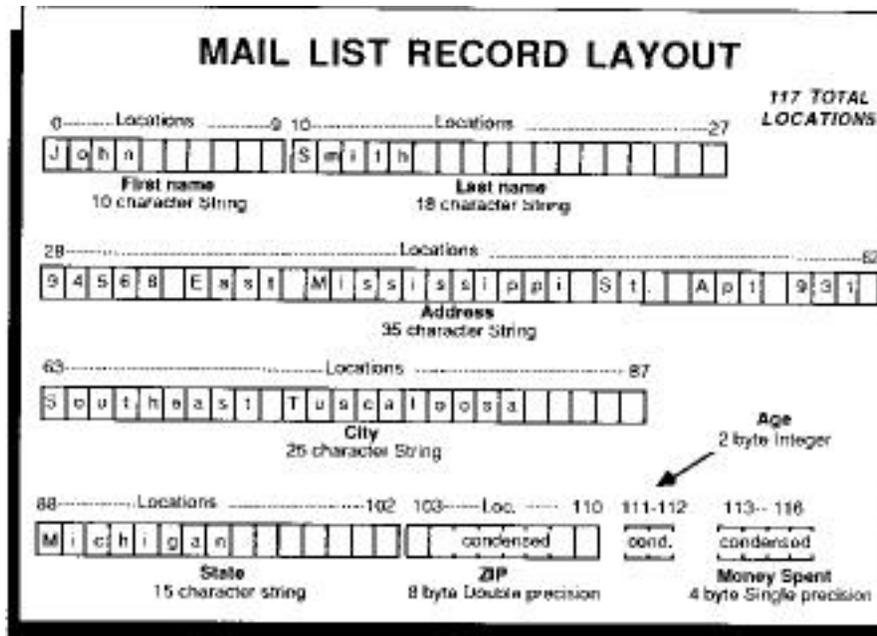
RANDOM METHOD

CREATING A MAIL LIST USING THE RANDOM ACCESS METHOD

This mail list uses: First and Last name, Address, City, State, Zip, Age and Money spent. The first thing to do is calculate the record length for the mail list file. This is done by calculating the space requirements for each field in a RECORD.

<u>FIELD</u>	<u>VARIABLE TYPE</u>	<u>SPACE NEEDED</u>
FIRST NAME	STRING\$	10 characters
LAST NAME	STRING\$	18 characters
ADDRESS	STRING\$	35 characters
CITY	STRING\$	25 characters
STATE	STRING\$	15 characters
ZIP	DOUBLE PRECISION	8 bytes (holds up to 14 digits)
AGE	INTEGER	2 bytes (Holds up to 32,767)
<u>MONEY SPENT</u>	<u>SINGLE PRECISION</u>	4 bytes (Holds up to 6 digits)
Totals:	8 VARIABLES	117 bytes per RECORD

The following illustration illustrates how the mail list data is stored within each RECORD. LOCATION numbers are shown by position.



RANDOM METHOD

MAIL LIST PROGRAM

The following program will READ# and WRITE# mail list data as described on previous pages. The names are input from the user and a mail list record is created for each name.

You will be able to retrieve, print, and search for names in the mail list and, with some simple modifications (like using the sort routines in the ARRAY section of this manual) you will have a complete mail list program ready to use.

EXPLANATIONS OF THE MAIL LIST PROGRAM BY LINE NUMBER

10-21	Asks if you want to create a new file. If you say yes the old data is written over.
22	If old data is being used, the data in RECORD zero is READ to find out how many names are on the disk. NR holds the number of records on the disk.
25-77	Puts a menu on the screen and awaits user input.
80	"END" routine. Closes file and exits the program.
100-210	"ADD" names to mail list. Gets data from user, checks if OK. If not OK starts over. Note that the spaces in the input statements are for looks only. Space may be omitted.
220	If not OK then redo the input.
230-255	Gets the disk record (DR) from NR. Saves the variables to disk, then increments the number of records. (NR=NR+1) and saves it to disk record zero.
500-590	PRINT(s) all the names in the file to the printer. (Change LPRINT to PRINT for screen output).
700-780	"FIND" all occurrences of LAST NAME or PART of a LAST NAME. To find all the names that start with "G" just type in "G". To find "SMITH" type in "SMITH" or "SMIT" or "SM".
1000-1040	"READ A MAIL LIST ITEM" READ(s) RECORD DR from the disk into the variables FIRST_NAME\$, LAST_NAME\$, ADDRESS\$, ...
1100-1140	"WRITE A MAIL LIST ITEM" WRITES the variables FIRST_NAME\$, LAST_NAME\$, ADDRESS\$, ... out to the RECORD specified by DR.

HINTS: Spaces are not important when typing in the program, except between double quotes (if you have set "Spaces required between keywords" they will be required).

RANDOM METHOD

MAIL LIST PROGRAM EXAMPLE

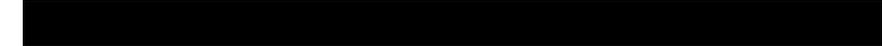
```
0010 CLS
0015 OPEN"R",1,"MAIL",117
0016 INPUT"CREATE A NEW FILE:Y/N";A$: IF A$><"Y" THEN 22
0021 NR=1: RECORD1,0: WRITE#1,NR:REM NR=Number of names in list
0022 RECORD 1,0: READ#1, NR
0025 DO: CLS
0030 PRINT"MAIL LIST PROGRAM"
0040 PRINT"1. Add names to list", "Number of names: ";NR-1
0050 PRINT"2. Print list"
0052 PRINT"3. Find names"
0055 PRINT"4. End"
0060 INPUT@ (0,7)"Number: ";ANSWER: IF ANSWER<1 OR ANSWER>4THEN60
0075 ON ANSWER GOSUB "ADD", "PRINT", "FIND"
0077 UNTIL ANSWER=4
0079 :
0080 "END": CLOSE#1: END
0099 :
0100 "ADD"
101 CLS
102 PRINT"MAIL LIST INPUT": PRINT
1030 INPUT"First Name: ";FIRST_NAME$
1040 INPUT"Last Name: ";LAST_NAME$
1050 INPUT"Address: ";ADDRESS$
1060 INPUT"City: ";CITY$
1070 INPUT"State: ";STATE$
1080 INPUT"ZIP: "ZIP#
1090 INPUT"AGE: ";AGE%
1100 INPUT"Money Spent:";SPENT!
1110 PRINT
1120 INPUT"Is everything correct? Y/N: ";A$: IFA$<>"Y"THEN "ADD"
1130 RECORD 1,0:READ#1,NR: DR=NR: NR=NR+1: REM NR is incremented
1140 GOSUB"WRITE A MAIL LIST ITEM": REM when names added
1150 RECORD 1,0: WRITE#1, NR : REM Stores records to record zero
1160 RETURN
1170 :
1180 :
1190 "PRINT"
1200 REM Change LPRINT to PRINT if screen output preferred
1210 RECORD 1,0: READ#1,NR
1220 FOR X=1TO NR-1: DR=X :REM DR=DISK RECORD
1230 GOSUB"READ A MAIL LIST ITEM"
1240 LPRINT FIRST_NAME$;" ";LAST_NAME$
1250 LPRINT ADDRESS$
1260 LPRINT CITY$;" ";STATE$;" ";ZIP#
1270 LPRINT AGE%, "SPENT:"; USING"$###,###.##";SPENT!
1280 LPRINT:IF FLAG=99 RETURN
1290 NEXT
1300 DELAY 3000
1310 RETURN
```

Continued next page

RANDOM METHOD

```
0700 "FIND"
0704 CLS
0705 RECORD 1,0:READ#1, NR
0710 IF NR=1 THEN PRINT "No names to find!":DELAY 999:RETURN
0720 INPUT"NAME TO FIND: ";F$:F$=UCASE$(F$)
0730 FOR X=1 TO NR-1
0740     DR= X: GOSUB"READ A MAIL LIST ITEM"
0750     T$=UCASE$(LAST_NAME$) :REM CASE must match
0755     IF INSTR(1,T$,F$) THEN FLAG=99: GOSUB 540: FLAG=0
0760 NEXT
0770 INPUT "LOOK FOR ANOTHER? Y/N:";A$:IFA$="Y" THEN 700
0780 RETURN
0781 :
0782 :
1000 "READ A MAIL LIST ITEM"
1001 REM:This routine READS RECORD DR
1020 RECORD 1, DR
1030 READ#1, FIRST_NAME$;10, LAST_NAME$;18, ADDRESS$;35,
1035 READ#1, CITY$;25, STATE$;15, ZIP#, AGE%, SPENT!
1040 RETURN
1041 :
1042 :
1100 "WRITE A MAIL LIST ITEM"
1101 REM: This routine WRITES RECORD DR
1110 REM CALL WITH DR=DISK RECORD NUMBER TO WRITE
1120 RECORD 1,DR
1130 WRITE#1, FIRST_NAME$;10, LAST_NAME$;18, ADDRESS$;35
1135 WRITE#1, CITY$;25, STATE$;15, ZIP#, AGE%, SPENT!
1140 RETURN: END
```

MIXING FILE METHODS



MIXING FILE METHODS



MIXING SEQUENTIAL AND RANDOM FILE METHODS

Since ZBasic stores data as a series of bytes whether sequential methods or random methods are used, these methods may be intermixed.

The following program uses both methods. The program reads files from the mail list program created with the random access method earlier in this chapter.

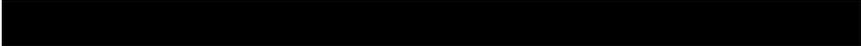
The second and third lines read the number of records in the file. Then the list is read off the disk sequentially using the DO/UNTIL loop.

To read and print the mail list in sequential order:

```
OPEN "I", 1 "MAIL", 117
RECORD 1.0:READ#1, NR:REM Gets a number of records to read
RECORD 1,1: REM Set the pointer to the first record
REM Change LPRINT to PRINT if screen output preferred
DO: NR=NR-1: REM Counts down the number of names
  READ#1, FIRST_NAME$;10, LAST_NAME$;18, ADDRESS$;35,
  CITY$;25, STATE$;15, ZIP#, AGE% SPENT!
  LPRINT FIRST_NAME$;" ";LAST_NAME$
  LPRINT ADDRESS$
  LPRINT CITY$;" ";STATE$;" ";ZIP#
  LPRINT AGE%, "SPENT: "; USING "$###,###.##";SPENT!
LPRINT
UNTIL NR=1: REM Until the last name is read
CLOSE#1
END
```

The READ#1 after the DO reads the data in. Whenever read or write functions are executed, ZBasic automatically positions the file pointer to the next position.

DISK ERRORS





DISK ERROR MESSAGES

If a disk error occurs while a program is running, ZBasic will print a message something like this:

```
File Not Found Error in File #02
(C)ontinue or (S)top?
```

If you type "S", ZBasic will stop execution of the program and return to the disk operating system (or to the editor if you are in interactive mode).

If you press "C", ZBasic will ignore the disk error and continue with the program. This could destroy disk data!!

The following pages will describe how to "TRAP" disk errors and interpret disk errors which may occur.

END OF FILE CHECKING

Some versions do not have an "END OF FILE" command because some operating systems do not have this capability. Example of END OF FILE checking for some versions:

```
ON ERROR GOSUB 65535: REM Set for User Error trapping
OPEN"1",1,"DEMO":IF ERROR PRINT ERRMSG$(ERROR):STOP
DO
  LINEINPUT#1,A$
UNTIL ERROR <>0
IF ERROR >< 257 THEN PRINT ERRMSG$(ERROR): STOP
REM 257=EOF Error in filename 1 (See error messages)
ERROR=0:REM You MUST reset the ERROR flag.
ON ERROR RETURN:REM Give error checking back to ZBasic
CLOSE#1
```

Note: Many versions have an EOF function. See your appendix for details.

DISK ERRORS

TRAPPING DISK ERRORS

ZBasic provides three functions for disk error trapping:

ON ERROR GOSUB 65535	Gives complete error trapping control to the user. User must check ERROR (if ERROR<>0 then a disk error has occurred) and take corrective action if any disk errors occur. (Remember to set ERROR=0 after a disk error occurs). ZBasic will not jump to a subroutine when the error occurs. The 65535 is just a dummy number. See the ON ERROR GOSUB line:
ON ERROR GOSUB line	GOSUB to the line number or label specified whenever and wherever, ZBasic encounters a disk error.
ON ERROR RETURN	Gives error handling control back to ZBasic. Disk error messages will be displayed if a disk error occurs.

When you are doing the ERROR trapping it is essential that ERROR be set to zero after an error is encountered (as in line#45 and #1025 in the program example). Failure to set ERROR=0 will cause additional disk errors.

DISK ERROR TRAPPING EXAMPLE

The following program checks to see if a certain data file is there. If disk error 259 occurs (File Not Found error for file #1), a message is printed to insert the correct diskette:

```
10 ON ERROR GOSUB "CHECK DISK ERROR"
15 REM Line above Jumps to line 1000 if any disk error occurs
20 OPEN"I",1,"TEST"
30 IF ERROR=0 THEN 50
40 INPUT"Insert Data diskette: press <ENTER>";A$
45 ERROR=0:REM You MUST reset ERROR to zero!
46 GOTO 20 :REM Check diskette again...
50 ON ERROR RETURN: REM ZBASIC DOES DISK ERROR MESSAGES NOW...
.
.
.
1000 "CHECK DISK ERROR"
1003 REM ERROR 259 is "File Not Found Error in File #01"
1005 IF ERROR=259 RETURN
1010 PRINT ERRMSG$(ERROR) :REM Prints error if not 259
1015 INPUT" (C)ont. or (S)top? ";A$
1020 A$=UCASE$(A$) : IFA$<>"C" THEN STOP
1025 ERROR=0:REM You MUST reset ERROR to zero!
1030 RETURN
```

Note: This method may not work on some Disk Operating Systems (like CP/M). Check your computer appendix for specifics.

DISK ERRORS

DISK ERROR CODES AND MESSAGES

If you wish to do the disk error trapping yourself (using ON ERROR GOSUB), ZBasic will return the ERROR CODE in the reserved variable word "ERROR".

For instance, if a "File not Found Error in file# 2" occurs, then ERROR will equal 515. To decode the values of 'ERROR', follow this table:

DISK ERROR CODES & MESSAGES

<u>ERROR</u>	<u>ERROR CODE</u>
No Error in File #	0
End of File Error in File #	1 (257=file#1, 513=file#2, 769=file#3, etc.)
Disk Full Error in File #	2
File Not Found Error in File #	3
File Not Open Error in File #	4
Bad File Name Error in File #	5
Bad File Number Error in File #	6
Write Only Error in File #	7
Read Only Error in File #	8
Disk Error in File #	9-255

ERROR CODE=ERROR AND 255
FILE NUMBER=ERROR>>8

ERROR FUNCTION

ERROR returns a number between 0 and 32,767. IF ERROR does not equal zero than a disk error has occurred. The disk error code of the value returned in ERROR is deciphered by using one of the following equations or statements:

IF ERROR =515 calculate the disk error type by:

ERROR AND 255 =3 File Not Found Error in File #

ERROR>>8 =2 File Number is 2

ERRMSG\$(ERROR)= File Not Found Error in File #02

Also See ERROR and ERRMSG\$ in the reference section.



Important Note: To avoid getting the same error again...ALWAYS set ERROR back to zero after an error occurs; ERROR=0.



Also see SYSERROR in the Macintosh appendix.

SCREEN AND PRINTER



SCREEN AND PRINTER



SCREEN AND PRINTER

ZBasic has several functions and commands for screen and printer control. PRINT or LPRINT are the most frequently used. The following syntax symbols are used to control the carriage return and TAB for either PRINT or LPRINT:

PRINT SYNTAX

Semi-Colon “;”

Comma “,”

DEF TAB=n

RESULT

Suppress Carriage return and linefeed after printing. subsequent prints will start at the cursor position.

TAB over to the next TAB stop. The default is 16: TAB stops are: 16, 32, 48, 64, ... 25 (also see DEF TAB below).

Defines the space between the TAB stops for comma (.). Any number from 1-255. If 10 is used then positions 10, 20, 30, ...250, are designated as TAB stops.

PRINT EXAMPLES

PRINT”HI”

PRINT “HI”;

PRINT “HI”,

PRINT TAB(20)”HI”

PRINT ,”HI”

PRINT USING”##.##”;23.2

POS(0)

POS(1)

RESULT

Screen PRINT “HI” to the current cursor position and move to the beginning of the next line. <CR>

Screen PRINT “HI” and DON’T move to next line (the semi-colon suppresses the carriage return)

Screen PRINT “HI” and move over to next TAB position.

Print “HI” at the 20th position over from the left or at the current position if past column 20.

Print “HI” at the next TAB stop position. See “ DEF TAB”.

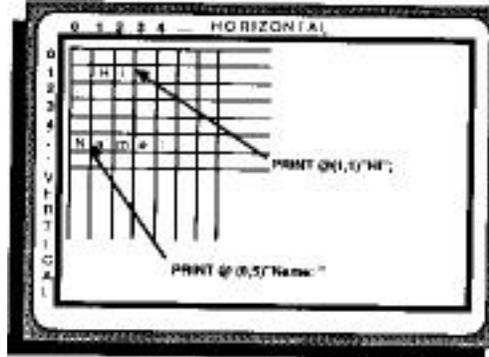
PRINTS 23.20 and moves to the next line. See “USING” in the reference section for further information.

Returns the horizontal cursor position on the screen where the next character will be printed.

Returns horizontal cursor position of the printer where the next character will be printed.

SCREEN AND PRINTER

PRINTING AT A SPECIFIC SCREEN LOCATION



`PRINT @(H,V)\"HI\"`

Start print H characters over horizontally and V lines down vertically from the upper left hand corner of the screen, then move to the beginning of the next line (Use a SEMI-COLON or COMMA to control the carriage return).

`PRINT %(Ghoriz,m Gvert)`

Position the print output to be at the graphics coordinates specified by Ghoriz, Gvert (or as close as possible for that computer. Great for easy porting of programs.

`CLS [ASCII]`

Fill Screen with spaces to the end of the LINE or to the end of the PAGE (screen).

`STRING$(Qty, ascii or string)`

Used to print STRINGS of characters. `STRING$(10,\"X\")` prints 10 X's to the current cursor position. `STRING$(25,32)` will print 25 spaces.

`SPACE$(n) or SPC(n)`

Prints n spaces from current cursor position.

`COLOR [=] n`

Sets the color of Graphics output and sometimes text. (0= background color, usually black. -1= foreground, usually white).*

`MODE [=] n`

Sets screen attributes. Some computers allow 80 character across or 40 characters across, etc.. Graphics may also be definable.*

`ROUTE byte integer`

Used to route output to the screen, printer or disk drive. *

* See Computer Appendix for specifics.

SCREEN AND PRINTER

PRINT %

The PRINT % command functions exactly the same way as PRINT @ except the X-Y coordinate specifies a screen graphic position instead of a character position.

Since ZBasic utilizes device independent graphics, this is a handy way of making sure the text goes to the same place on the screen regardless of the computer being used.

Use MODE to set certain character styles for some computers.

Examples:

PRINT % (512, 383)	Print to middle of screen
PRINT % (0,0)	Upper left corner of screen
PRINT % (0,767)	Lower left corner of screen



Same as the toolbox MOVETO function. ZBasic coordinates unless COORDINATE WINDOW is used.

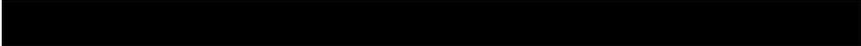
TYPICAL VIDEO CHARACTER LAYOUTS

Here are some of the typical character layouts for a few of the more popular computers:

COMPUTER	Columns (across)	Rows (down)
IBM PC and compatible	80 or 40	25
APPLE //E, //C	80 or 40	24
TRS-80 Model I, III	64 or 32	16
TRS-80 Model 4, 4p	80 or 40*	24
CP/M-80 computers	80	24
Macintosh	Almost anything...	See appendix

*Will also run TRS-80 models 1,3 version.

KEYBOARD INPUT



KEYBOARD INPUT



KEYBOARD INPUT

ZBasic utilizes the INPUT and LINEINPUT statements of getting keyboard data from a user. There are many options allowed so that input may be configured for most input types. Parameters may be used together or by themselves in any order. Syntax for INPUT and LINEINPUT:

[LINE]INPUT[:][[@or%] (horiz,vert);] [!] [& n,] ["string constant";] ar [, var[,...]]

LINEINPUT	Optional use if INPUT. Allows inputting quotes, commas, and some control characters.
;	A semi-colon directly following "INPUT" disables the carriage return (cursor stays on same line after input).
& n,	"&" directly following "INPUT" or semi-colon, sets the limit of input characters to n. Length of strings used in INPUT must be one greater than n.
!	An exclamation point used with "&" terminates the INPUT when the character limit, defined by "&" , is reached, without pressing <ENTER>. If "!" is not used, <ENTER> ends input.
@(horiz, vert);	Positions the INPUT message to appear at character coordinates horiz characters over & vert lines down.
%(horiz, vert);	Positions the INPUT message to appear at the closest graphic coordinates horiz pixels over & vert pixels down.
"string constant";	Prints a message in front of the input.
var [,var][,...]	The variable(s) to receive the input. Using more than one variable at a time is allowed except with LINEINPUT.



Important Note: When using strings with INPUT make sure that you define the length of the string at least one character more than will be input.

KEYBOARD INPUT

EXAMPLES OF REGULAR INPUT

<u>EXAMPLE</u>	<u>RESULT</u>
INPUT A\$	Wait for input from the keyboard and store the input in A\$. Quotes, commas and control characters cannot be input. <ENTER> to finish. A carriage return is generated when input is finished (cursor moves to beginning of next line).
INPUT"NAME: ";A\$	Prints "NAME: " before input. A semi-colon must follow the last quote. A carriage return is generated after input (cursor moves to next line).
INPUT;A\$	Same as INPUT A\$ above, only the semi-colon directly after INPUT disables the carriage return (cursor stays on the same line).

EXAMPLES OF LIMITING THE NUMBER OF CHARACTERS WITH INPUT

<u>EXAMPLE</u>	<u>RESULT</u>
INPUT &10, A\$	Same as INPUT A\$ only a maximum of ten characters may be input. (&10) A carriage return is generated after input (cursor moves to the beginning of the next line). The limit of input is set for ALL variables, not each.
INPUT ;&10, I%	Same as INPUT &10, except the SEMI-COLON following INPUT stops the carriage return (cursor stays on line).
INPUT !&10, A\$	Same as INPUT & 10 except INPUT is terminated as soon as 10 characters are typed (or <ENTER> is pressed).
INPUT ;!&10, "NAME: ";A\$	Same as INPUT ;&10,A\$ except no carriage return is generated (semi-colon). INPUT is terminated after 10 characters(&10 and Exclamation pint). and the message "NAME: " is printed first.
LINEINPUT;!&5,"NAME: ";A\$	LINEINPUT A\$ until 5 characters or <ENTER> is pressed. (no carriage return after <ENTER> or after the 5 characters are input. Accepts commas and quotes.)

Note 1: Wherever INPUT is used, LINEINPUT may be substituted when commas, quotes or some other control characters need to be input (except with multiple variables).

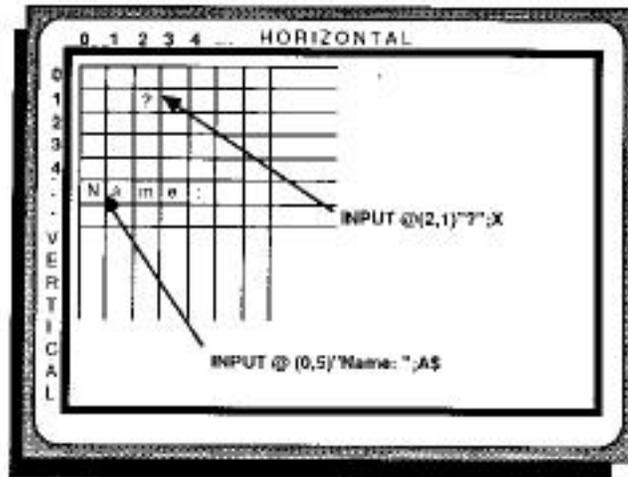
Note 2: If more than one variable in INPUT, commas must be included from the user to separate input. If all the variables are not input, the value of those variables will be null.



In certain cases EDIT FIELD, MENU or BUTTON may be preferable. See appendix.

KEYBOARD INPUT

INPUTTING FROM A SPECIFIC SCREEN LOCATION



`INPUT @(H,V); A$`

Wait for input at TEXT screen POSITION defined by Horizontal and Vertical coordinates. No "?" is printed. A carriage return is generated.

`INPUT %(gH,gV);A$`

Input from a graphic coordinate. Syntax is the same as "@". Very useful for maintaining portability without having to worry about different screen widths or character spacing.

`INPUT@(H,V);!10,"AMT: ";D#`

the next line).

Prints "AMT:" at screen position H characters over by V characters down. D# is input until 10 characters, or <ENTER>, are typed in, and the input is terminated without generating a carriage return (the cursor DOES NOT go to the beginning of

`INPUT%(H,V);!10,"AMT: ";D#`

Prints "AMT:" at Graphic position H positions over by V positions down. D# is input until 10 characters, or <ENTER>, are typed in, and the input is terminated without generating a carriage return (the cursor DOES NOT go to the beginning of the next line).

Note: Replace INPUT with LINEINPUT whenever there is a need to input quotes, commas and control characters (except with multiple variables).

KEYBOARD INPUT

INPUT %

The INPUT % command functions exactly the same way as INPUT@ except the X-Y coordinate specifies a screen graphic position instead of a character position.

Since ZBasic utilizes device independent graphics, this is a handy way of making sure the INPUT goes to the same place on the screen regardless of the computer being used.

Use MODE to set certain character styles for some computers.

Examples:

INPUT%(512, 383) middle of screen
INPUT%(0,0) upper left corner of screen
INPUT%(0,767) lower left corner of screen



Although all parameters above function properly, EDIT FIELD, MENU or BUTTON are preferable for getting user input. See appendix.

TYPICAL VIDEO CHARACTER LAYOUTS

Here are some of the typical character layouts for a few of the more popular computers:

COMPUTER	<u>Columns (across)</u>	<u>Rows (down)</u>
IBM PC and compatible	80 or 40	25
APPLE // series	80 or 40	24
TRS-80 Model I, III	64 or 32	16
TRS-80 Model 4, 4p	80 or 40	24
CP/M-80 computers	80	24
Macintosh	Almost anything...	See appendix

KEYBOARD INPUT

INKEY\$

Unlike INPUT which must WAIT for characters, INKEY\$ can receive characters from the keyboard "on the fly". When INKEY\$ is encountered in a program, the keyboard buffer is checked to see if a key has been pressed. For computers with no buffer, the keyboard is checked when the command is encountered. If a key is pressed, INKEY\$ returns the key. If no key has been pressed, INKEY\$ returns a null string. Examples:

I\$=INKEY\$

When the program reaches the line with this command on it, ZBasic checks to see if a character is in the input buffer. If a key has been pressed it will be returned in I\$. Otherwise I\$ will contain nothing (I\$ will equal "" or LEN(I\$)=zero).

IF INKEY\$="S" STOP

If the capital "S" key is pressed the program will stop. Sometimes more appropriate than using TRONB or TRONX for debugging purposes.

DO: UNTIL LEN(INKEY\$)
DO: UNTIL LEN(INKEY\$)=0

Wait for any key press, then continue
Clears characters out of INKEY\$ buffer

Note: TRONX, TRON or TRONB may cause INKEY\$ to function improperly!



Macintosh: If doing EVENT Trapping or any TRON type, the INKEY\$ function may operate incorrectly. Use DIALOG(16) instead. See appendix for examples. **MSDOS:** See appendix for special ways of getting function keys (INKEY\$ returns two characters).

INKEY\$ EXAMPLE

The program below will wait awhile for a key to be pressed. If you make it wait to long, it will complain loudly. If you do press a key, it will tell you which key was pressed. If you press "S" or "s", the program will stop.

```
"Start": CLS
DO
  A$INKEY$:REM Check if a key has been pressed
  X=X+1: IF X>3000 THEN GOSUB"YELL FOR INPUT!":REM Timer
UNTIL LEN(A$): REM If a key is pressed then LEN(A$)=1
PRINT "You pressed ";A$
X=0: REM Reset timer
IF A$="S" OR A$="s" THEN STOP: REM PRESS "S" to STOP!
GOTO "Start":REM Go look for another key
:
"YELL FOR INPUT!": REM This routine complains
PRINT"HURRY UP AND PRESS A KEY! I'M TIRED OF WAITING"
X=0:REM Reset Timer
RETURN
```

LOOPS





LOOPS

Loops are sections of a program that repeat over and over again until a condition is met.

Loops are used to make programs easier to read by avoiding IF THEN and GOTO, (although these commands may also be used to loop). ZBasic has a number of ways of looping or executing a routine until a condition is met.

* **FOR, NEXT, STEP**
 * **DO, UNTIL**
 WHILE, WEND

* Each of these loop types is executed at least once.

ENTERING OR EXITING LOOPS

ZBasic loops may be entered or exited without ill affects. Some compilers require you to use a loop EXIT statement. This is not required with ZBasic. Just use a GOTO or RETURN to exit as appropriate.

IMPORTANT LOOP REQUIREMENTS

ZBasic requires that each FOR has one, and only one, NEXT. Each WHILE must have one WEND and each DO must have one UNTIL. Otherwise a STRUCTURE error will result when you attempt to RUN the program.

AUTOMATIC INDENTING OF LOOPS

ZBasic automatically indents loops two characters in listings for readability (LIST).

LOOPS

FOR-TO-STEP NEXT

```
FOR VAR counter= start expression TO end expression [STEP expression]
  Program flow...
NEXT [VAR counter]
```

STEP is an optional part of FOR/NEXT. If STEP is omitted, the step is one. An example of a FOR-NEXT-STEP loop:

```
FOR X=0 TO 20 STEP 2
  PRINT X;
NEXT X
program continues...
```

LINE 1: Begin the loop where X is incremented in STEPs of 2 (0,2,4,6...20)

LINE 2: Prints the value of X each time the loop is executed.

LINE 3: If X => 20 the loop falls through to line 4. X will equal 22 in line 4 of this example program.



FOR-NEXT loops will go through the loop at least once regardless of the values in the FOR instruction. See WHILE-WEND for immediate exiting.

To count backwards in a FOR/NEXT loop set STEP to a negative number.

Note 1: STEP zero will cause an endless loop.

*Note 2: With integer loops, be sure the maximum number is less than 32,767; otherwise an endless loop may occur for some systems. The reason for this is that the sign of the number increments to -32768 after 32767 which restarts the loop all over again! Endless loop example:

```
FOR X%= 1 TO 32767 <--Endless loop!
NEXT X%
```

Note 3: STEP number must stay within the integer range. STEP 32767 would create an endless loop.

Note 4: Unlike most other languages, FOR-NEXT loops may be entered or exited in the middle with no ill effects.



*The same problem arises with four byte integers when the maximum LongInteger number in the FOR loop exceeds 2,147,483,647.

DO UNTIL

DO
Program flow...
UNTIL conditional expression is TRUE

```
DO
  X=X+2
  PRINT X;
UNTIL X>19
program continues...
```

LINE 1: Start of the DO loop
LINE 2: Make X=X+2
LINE 3: PRINT the value of X each time the loop is executed.
LINE 4: If X<20 then go back to the beginning of the loop. When X>19 program falls through to the next statement (line 4 in example)



A DO loop will execute at least once. In contrast to WHILE-WEND, which checks the condition at the beginning of the loop, DO-UNTIL checks the condition at the end of the loop. Use WHILE-WEND when you need to check the condition at the beginning.

Note: Unlike most other languages, the loop may be entered or exited in the middle with no ill effects. For instance, in line 2 above, you could use: IF X>10 then RETURN. This would not cause any problems in the program.

LOOPS

WHILE WEND

WHILE conditional expression
Program flow...
WEND end loop here when condition of WHILE is FALSE

```
        WHILE X<20
            X=X+2
            PRINT X;
        WEND
program continues...
```

LINE 1: Continue the loop while X is less than 20.
LINE 2: Make X=X+2
LINE 3: Print the value of X each time the loop is executed.
LINE 4: If X is less than or equal 20 then go back to the WHILE and do the loop again, otherwise continues at the first statement after WEND.



In contrast to DO-UNTIL and FOR-NEXT (which check the condition at the end of a loop), **WHILE-WEND checks the condition at the beginning of the loop and will exit immediately if the condition is not met.**

Note: Unlike most other languages, a WHILE-WEND loop may be entered or exited in the middle with no ill effects. For instance, in line 30 above, you could have used: IF X>10 then RETURN. This would not cause any problems in the program.

FUNCTIONS AND SUBROUTINES



FUNCTIONS AND SUBROUTINES

ZBasic contains some powerful tools for creating re-usable subroutines and appending or inserting them into other ZBasic programs that you create.

APPEND

APPEND is a command that will take an un-line numbered subroutine and insert it anywhere in an existing program. The syntax for the command is APPEND line number or label, filespec.

To save a subroutine or program without line numbers, use the SAVE+ command. MERGE is available for merging subroutines or programs with line numbers into an existing program.

DEF FN

Zbasic incorporates the DEF FN and FN statements similar to many other BASIC languages. This is very handy for creating functions that may be used like commands in a program.

A function is given a name and may be called and passed variables. FN's save program space. Note that functions may utilize other functions within definitions and program code.

Examples of using DEF FN to create Derived Math functions.

```
DEF FN e# = EXP(1.)
DEF FN Pi# = ATN(1) << 2
DEF FN SEC# (X#) = 1. \ COS (X#)
DEF FN ArcSin# (X#) = ATN (X# \ SQR(1-X# * X#))
DEF FN ArcCos#(X#) = ATN(1.)*2-FN ArcSin# (X#)
```

Examples of program use:

```
PRINT FN Pi#
Angle# = SIN (FN ArcSin#(I#))
PRINT FN ArcCos#(G#)
```

Note: Be sure to define the function at the beginning of the program before attempting to use it otherwise an UN DEF error will result at compile time.

FUNCTIONS AND SUBROUTINES

LONG FN

Included is a sophisticated and powerful multiple line function called LONG FN.

LONG FN allows you to create multi-line functions as large as a subroutine and allows you to pass variables to the routine. This comes in very handy for creating reusable subroutines that you can insert or APPEND to other programs.

LONG FN is similar to DEF FN except that the function being defined may be many lines long. Use END FN to end the LONG FN subroutine. WARNING: Do not exit a LONG FN except at END FN otherwise system errors may result.

Example of LONG FN to remove trailing spaces from a string:

```
LONG FN Remove trailing spaces from a string:
  WHILE ASC(RIGHT$(x$,1))=32
    x$= LEFT$(x$, LEN(x$)-1)
  WEND
END FN= x$
Name$="ANDY      "
PRINT X$, FN RemoveSpace$(Name$)
z$=FN RemoveSpace$(fred$)
```

Example of a LONG FN for doing a simple matrix multiplication:

```
DIM A%(1000)
LONG FN MatrixMult%(number%, last%)
  FOR temp%= 0 to last%
    A%(temp%)=A%(temp%)*number%
  NEXT
END FN
A% (0)=1: A%(1)=2:A%(2)=3
FN MatrixMult%(10,3)
PRINT A%(0), A%(1), A%(2)
```

SYNTAX OF DEF FN AND LONG FN NAMES

FN names have the same syntax as variable names. A function that returns a string value should end with a \$. A function that returns a double precision value should end with a #.

AUTOMATIC INDENTATION

ZBasic automatically indents that code between a LONG FN and END FN so programs are easier to read.

SAVING FUNCTIONS FOR USE IN OTHER PROGRAMS

To save DEF FN'S or LONG FN's (or any subroutine) for future use, use SAVE+. This saves the subroutine without line numbers so it may be used in other programs by loading with the APPEND command (be sure to avoid line number references and GOTOs in subroutines to make them easily portable).

FUNCTIONS AND SUBROUTINES

MORE EXAMPLES OF LONG FN

The following example will check to see if a random file specified by the filename file\$ exists. If it does it will open it as a random file. If it does not exist, it will return a disk error.

Remember; with OPEN"R" if the file exists it is opened, if it doesn't exist it is created. You may not want it created in certain circumstances (like if the wrong diskette is in a drive).

```
LONG FN Openfile%(files$, filenum%, reclen%)
  ON ERROR 65535: REM Disk error trapping on
  "Open file"
  OPEN"I",filenum%,file$
  LONG IF ERROR
    LONG IF (ERROR AND 255) <>3
      PRINT@(0,0);"Could not find ";file$;" Check disk drive"
      INPUT"and press <ENTER> when ready";temp%
      ERROR=0: GOTO "Open file"
    END IF
  XELSE
    CLOSE# filenum%
  END IF
  ON ERROR RETURN: REM Give error checking back to ZBasic
  OPEN"R",filenum%, file$, reclen%
END FN
```

EASY GETKEY FUNCTION

```
LONG FN GetKey$(Key$)
  DO
    Key$=INKEY$
  UNTIL LEN(Key$)
END FN - Key$
```

MACHINE LANGUAGE SUPPORT



MACHINE LANGUAGE SUPPORT



MACHINE LANGUAGE

Occasionally it is important to be able to use machine language programs with your program, whether for speed or to utilize special features of the hardware of that machine. ZBasic incorporates a number of special commands to integrate machine language subroutines into your programs.



CAUTION: Unless you have a working knowledge of the machine language of the source computer and target computer, use extreme caution when porting programs with machine language commands or subroutines.

MACHLG

This statement allows you to put bytes or words directly into your program:

```
CALL LINE "Machlg": END
"Machlg": REM EXAMPLE ONLY--> DO NOT USE!
MACHLG 10, 23, 233, 12, 0, B%, A, 34, 12, &EF
MACHLG 23, 123, 222, 123, 2332, GameScore%, &AA
```

Hex, Binary, Octal or Decimal constants, Integer variables, or VARPTR may be used. Be sure to put a machine language RETURN at the end of the routine if using CALL. Be sure you understand the machine language of your computer before using this command.

LINE

This gives you the address of a specific line as it appears in the object code. This allows you to CALL machine language programs starting at specific line numbers or labels. Syntax is

LINE label
or LINE line number



Since the Macintosh is a 16 bit machine, MACHLG code is stored in WORDS not BYTES. The code above would be stored in every other byte. With LINE parentheses are required because it is also a toolbox call i.e. LINE (n).

MACHINE LANGUAGE SUPPORT

CALL

Allows you to CALL a machine language program. The syntax is:

`CALL address`

Be sure the routine being called has a RETURN as the last statement if you wish to return control to your program.

If you wish to CALL a machine language subroutine in your program that was made with MACHLG, use CALL LINE *line number* or *label*.



These versions have additional parameter passing capabilities. See appropriate appendix under CALL for specifics.



The ProDOS version provides a special interface to the ProDOS Machine Language Interface (MLI). See appendix for specifics.

DEFUSR 0 - 9

Allows you to set up to 10 different machine language user routines. The syntax for using this statement is:

`DEFUSR digit =address`

This command may be used to pass parameters or registers. See your computer appendix for the specifics about your computer. There are also default routines. See USR in the reference section.

INTEGER BASE CONVERSIONS

ZBasic makes integer BASE conversions simple. Some of the commands for converting between BASED:

`BIN$, &X` `UNS$`
`HEX$, &H or &` `OCT$, &O`

See "Numeric Conversions" for specifics.



See DEFSTR LONG for configuring conversions above for LongInteger (and also CVI and MKI\$).

MACHINE LANGUAGE SUPPORT

OTHER MACHINE LANGUAGE COMMANDS

Other tools for machine language programmers include powerful PEEK and POKE statements that can work with 8, 16 or 32 bit numbers and BOOLEAN MATH

PEEK, POKE

In addition to the "standard" BYTE PEEK and POKE provided by many versions of BASIC, WORD (16 bit) and LONG (*32 bit) PEEK and POKE are also provided:

PEEK	8 BIT	POKE	8 BIT
PEEKWORD	16 BIT	POKEWORD	16 BIT
PEEKLONG	*32 BIT	POKELONG	*32 BIT



* Macintosh only at this time.

BINARY/BOOLEAN MATH FUNCTIONS

OR AND
XOR NOT
SHIFT LEFT SHIFT RIGHT

EXP and IMP may be emulated easily. See "Logical Operators" in the "Math" section of the manual.

VARIABLE POINTER

VARPTR (variable) will return the address of that variable.



Macintosh: Remember to use LongIntegers to store the address since Macintosh memory exceeds 65,535 (the limit of regular integers). Also see DEFSTR LONG for defining integer functions to do LongInteger. MSDOS: Check appendix for way of determining SEG of variable.

STRUCTURED PROGRAMMING



STRUCTURED PROGRAMMING



STRUCTURE

Much has been said about the difficulty of reading BASIC programs and the so-called spaghetti code created (the program flow is said to resemble the convoluted intertwinings of string spaghetti).

While we believe structure is important, we don't believe that a language should dictate how a person should compose a program. This inhibits creativity and may even paint programmers into corners.

Nevertheless, we have provided powerful structure support in ZBasic.

THAT NASTY "GOTO" STATEMENT

The GOTO statement has been classified by many as a programmer's nightmare. If you want programs that are easy to read, do not use this command. If you must use GOTO, do not use line numbers, use labels to make the code easier to follow.

LINE NUMBERS VERSUS LABELS

The standard line editor (command mode) uses line number for three reasons:

1. Remain compatible with older versions of BASIC
2. For the Standard line editor commands
3. To give more easily understandable error messages

To make programs easier to read you should use alphanumeric labels for subroutines or any other area of a program that does a specific function.

It is much easier to follow the flow of a program if GOSUB, GOTO and other branching statements use labels instead of line numbers.

To LIST programs without line numbers use LIST+. Many versions of ZBasic now use full screen editors that don't require line numbers. See your appendix for specifics.

STRUCTURED PROGRAMMING

INDENTATION OF LOOPS, LONG FN and LONG IF

Some versions of structured languages require that you manually indent nested statements for readability.

ZBasic does all the indenting automatically!

Each nested portion of a program will be indented 2 spaces when the program is listed. Program statements like FOR-NEXT, WHILE-WEND, DO-UNTIL, LONG FN, LONG-IF etc. will be indented.

Example using LIST+:

```
LONG FN KillFile(file$)
  PRINT@(0,10);"Erase ";file$;" Y/N";
  DO
    temp$=INKEY$
  UNTIL LEN(temp$)
  LONG IF temp$="y" or temp$="Y"
    KILL temp$
  END IF
END FN
FOR X=1 TO 100
  DO : G=G+1
    WHILE X<95
      PRINT "HELLO"
      LONG IF J< 4
        J=J+1
      END IF
    WEND
  UNTIL G >= 3.5
NEXT X
```

MULTIPLE LINE STATEMENTS

ZBasic allows putting more than one statement on a line with ":" (colon). While this is handy for many reasons, over-use of this capability can make a program line very difficult to understand.

UNSTRUCTURED 10*FORX=1TO100:DO:G=G+1:PRINT G:UNTILG=99:NEXT

STRUCTURED FOR X = 1 TO 100
DO : G=G+1
PRINT G
FOR V=1 TO 20:NEXT
UNTIL G=99
NEXT X
*FOR V=1 TO 20:NEXT

*Note: An asterisk will appear at the beginning of a line containing a complete loop if that line is not already indented. In that case the line will be un-indented two spaces (as in the examples above).

STRUCTURED PROGRAMMING

SPACES BETWEEN WORDS

To make code more readable, you should insert spaces between words, variables and commands, just as you do when writing in English. While ZBasic does not care if spaces are used (unless you configure ZBasic to require spaces), it is a good practice to insert spaces at appropriate places to make reading the program easier.

Hard to Read	<code>IFX=93*24THENGOSUB"SUB56"ELSEEND</code>
Easier to Read	<code>IF X=93*24 THEN GOSUB "SUB56" ELSE END</code>

VARIABLE NAMES

To make code more readable, use logical words for variables.

Hard to Read	<code>B=OP+I</code>
Easier to Read	<code>Balance = Old_Principle + Interest</code>

ZBasic allows variable name lengths up to the length of a line, but only the first 15 characters in the name are significant. Do not use spaces or symbols to separate words in a name, use underlines; `Building_Principle`, `Freds_House`.

Keywords may not be used in variable names unless they are in lowercase and "Convert to Uppercase" is "NO" (this is the default). Also see next paragraph.

INCLUDING KEYWORDS IN VARIABLES

To allow keyword in variables configure ZBasic for; "Spaces Required after Keywords" (not available on all systems). See "Configure".

HOW CASE AFFECTS VARIABLE NAMES

To make the variable "FRED" and "fred" the same variable configure ZBasic for "Convert to Uppercase". See "Configure".

GLOBAL VERSUS LOCAL VARIABLES

Programmers familiar with LOCAL variables in PASCAL and some other languages can structure their variable names to approximate this in ZBasic. (all ZBasic variables are global.)

GLOBAL variables should start with a capital letter.

LOCAL variables should start with lowercase. Many programmers also use (and re-use) variables like `temp$` or `local$` for local variables.

STRUCTURED PROGRAMMING

DEFINING FUNCTIONS

Use DEF FN or LONG FN to define functions and then call that function by name. This is easy reading for people trying to decipher your programs. It saves program space as well. FN names have the same definition as variable names. Passing values to functions in variables is also very easy.

LONG FN may be used when a function the size of a subroutine is needed. One FN may call previously defined functions.

LOADING PREVIOUSLY CREATED SUBROUTINES

To insert subroutines you have used in previous programs, use the APPEND command. This will append (or insert) a program saved with SAVE+ (a non-line numbered program or subroutine), into the current program starting at the line number you specify; APPEND linenumber or label filename

Be sure to avoid the use of line numbers or GOTO statements in your subroutine to make them more easily portable.

If using variables that are to be considered LOCAL, we recommend keeping those variables all lowercase characters to differentiate them for GLOBAL variables (all ZBasic variables are GLOBAL).

Sometimes LONG FN may be more appropriate for re-usable subroutines.

LISTINGS WITHOUT LINE NUMBERS

To make program listings easier to read, use LIST+ or LLIST+ to list a program without line numbers.

ZBasic automatically indents nested statements with LIST for even more readability.



Macintosh: Listings can be sent to the Screen, LaserWriter or ImageWriter without line numbers and with keywords boldfaced by using LLIST+*.

MSDOS: Screen listings with highlighted keywords and no line numbers are accomplished with LIST+* (no printer support for highlighted keywords).

STRUCTURED PROGRAMMING

LONG IF

For more control of the IF statement, ZBasic provides LONG IF for improved readability and power.

UNSTRUCTURED

```
10 IIFX=ZTHENY=10+H:G=G+Y:F=F+RELSEGOSUB122:T=T-1
```

STRUCTURED

```
LONG IF X=Z
    Y=10+H
    G=G+Y
    F=F+R
XELSE
    GOSUB"READ"
    T=T-1
END IF
```

UNSTRUCTURED

```
10 FORI=-3TO3:PRINT"I= ";I:IF I> THEN IF I>-3 AND I<3
PRINT I;">0",ELSEPRINT"Inner If False":GOTO 30
20 *PRINT I;"<=0",:X=-4:DO:X=X+1:PRINT"X=";X:UNTILX=I
30 NEXT I
```

STRUCTURED

```
FOR I = -3 TO 3: PRINT "I= ";I
LONG IF I > 0
    LONG IF I > -3 AND I < 3
        PRINT I;"> 0",
    XELSE
        PRINT "Inner LONG IF false"
    END IF
XELSE
    PRINT I;"<= 0",
    X = -4
    DO X=X+1
        PRINT"X=";X
    UNTIL X=I
END IF
NEXT I
```

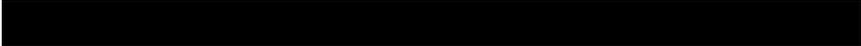


Important Note: Any loops enclosed in LONG IF structures must be completely contained with the LONG IF, END IF construct.



The Macintosh and IBM versions also support SELECT CASE, a structured, multi-conditional LONG IF statement. See appendices for syntax.

DEBUGGING TOOLS





DEBUGGING TOOLS

To get programs running bug-free in the shortest amount of time, ZBasic has incorporated some powerful error catching tools.

TRON Display program flow

Turns on the line trace statement. As the program is running, ZBasic will display the line number where the program is being executed on the screen.



Also see TRON 128 for sending the line numbers to the printer so the display is not affected.

TRONS Single Step

SINGLE STEP line trace debugging. Allows you to single step through that part of a program. To activate single step mode press CTRL Z. To single step press any key. To return to regular mode press CTRL Z again. To single step and display line numbers use TRONS:TRON. Note: CTRL S and CTRL Z will function during any TRON type.

TRONB Check for <BREAK> key

Sets a break point on that line and all the following lines of that program (until a TROFF is encountered). As each line is executed, the program will check if CTRL C or <BREAK> is being pressed.

If <BREAK> is pressed, the program will return to the edit mode (the operating system if RUN* was used). Without a break point the program will not respond to the <BREAK> key. No line numbers are displayed unless TRON was also used.



BREAK ON is often preferable as a check for <COMMAND PERIOD>. See appendix.

DEBUGGING TOOLS

TRONX Check for <BREAK> on that line only

Sets a break point only on that line. If CTRL C or <BREAK> is pressed as that line is executed, the program will return to the edit mode (if interactive) or to the operating system.

TROFF Disable all TRON modes

Turns off TRON, TRONB, TRONX and TRONS. Line number display and <BREAK> points will be disabled in the program flow following this statement.

ARRAY BOUNDS CHECKING

Set "Check Array Bounds" to "YES" when configuring ZBasic to make sure you do not exceed DIM limits. This is a RUN TIME error check and is very important for use during the debug phase.

Exceeding array limits could cause overwriting of other variables and faulty data.

After you have finished debugging your program, disable this function since it will slow execution speed and increase program size.

STRING LENGTH CHECKING (not all versions; check your appendix)

Set "String Length Checking" to "YES" when configuring ZBasic to make sure you do not exceed defined string length limits. This is a RUN TIME error check and is very important for use during the debug phase.

Exceeding string lengths could cause overwriting of other variables and/or faulty data.

After you have finished debugging your program, you may wish to disable this function since it will slow execution speed and increase program size.

COMPILE TIME ERROR CHECKING

ZBasic compile time error messages help you pinpoint the cause of the problem immediately by highlighting the error on the line and printing a descriptive message instead of an error number.

Unlike BASIC interpreters, ZBasic will not execute a program with syntax errors in it. If the program compiles without an error you can be sure it is at least free of syntax errors.

DISK ERROR CHECKING

ZBasic gives the programmer a choice of trapping disk errors themselves or letting ZBasic display the disk error. See "Disk Error Trapping" for more information.

PORTING PROGRAMS



PORTING PROGRAMS

Porting means taking a program from one computer and moving it to another computer of different type or model. As from an Apple to an IBM.

Because most ZBasic commands contained in the reference section of this manual (except USR, OUT, INP, PEEK, POKE, VARPTR, CALL and MACHLG) function the same way, it is very easy to move the source code from one machine to another.

The following pages will describe some of the problems and solutions of porting programs.

OBJECT CODE AND SOURCE CODE

There are two separate types of programs created with ZBasic and you should understand the differences.

SOURCE CODE This is the text part of a program you type into the computer and looks like the commands and functions you see in this manual. In order to turn SOURCE CODE into OBJECT CODE, ZBasic compiles it when you type RUN (or RUN* or RUN+).

OBJECT CODE The OBJECT CODE is what ZBasic creates from the SOURCE CODE after you type RUN. Object code is specific to a certain machine. i.e. an IBM PC uses an 8088 CPU and an Apple // uses a 6502 CPU. The ZBASIC OBJECT CODE for each of these machines is different and cannot be ported. Port the SOURCE CODE to the target machine and then recompile it into the OBJECT CODE of that computer.

FILE COMMANDS

ZBasic file commands work almost exactly the same way from one computer to the next. The areas to be aware of when porting code from one machine to another are covered in the following two paragraphs.

PORTING PROGRAMS

DISK CAPACITIES

Make sure the target machine has enough storage space to accommodate the program and program files being ported.

COMMON DRIVE CAPACITIES

IBM PC, XT, jr.	5.25"	320K-360K
	3.50"	780K
IBM PC AT	5.25	360K
	3.50"	780K
	variable density	1200k
Apple // series	5.25"	143k
	3.50"	800k
Macintosh	single sided	400K
Macintosh Plus	double sided	800K
Other:		
SSSD	5.25"	80K
SSSD	8.00"	200-500K
SSDD	5.25"	160K
DSDD	5.25"	320K
DSDD	8.00"	600-2000K

SSSD: Single sided Single density
SSDD: Single sided Double density
DSDD: Double sided Double density

FILESPECS

ZBasic filenames/filespecs work within the limitations of the disk operating system. When porting programs make sure the filespecs are corrected. For instance; if porting a program from a TRS-80 Model 3 to an IBM PC, you must change all references to a file like; FRED:1 to A:fred

Some computers cannot do RENAME or EOF. Others are incapable of certain DISK ERRORS. Be sure to study the DOS manual of the target machine for variations.

MEMORY

Memory is another area of importance when porting programs from one machine to another.

Porting from smaller machines to machines with larger memory should not be a problem, as long as other hardware is similar. Programs from TRS-80 MODEL I, III, 4, Apple //e and //c and CP/M 80 machines should port over to an IBM PC or Macintosh with little or no changes.

Porting a large program (128K or more) from a larger machine like an IBM PC or Macintosh to a smaller machine will require a number of memory saving measures covered in the following paragraphs:

CHAINING PROGRAMS TOGETHER

If a 128K program is being moved to a 64K system, you will have to split it up into two or more separate programs and CHAIN them together. Since ZBasic allows sharing variables while chaining, this should solve most problems.

PORTING PROGRAMS

CHECK STRING MEMORY ALLOCATION

ZBasic allows the user to change the size of strings. Since some programmers on larger machines may not be concerned with creating efficient code or keeping variable memory use down, check if string size has been set. Setting string size from the 256 byte default to 32 or 64 will reduce string variable memory requirements dramatically.

See DEFLEN, DIM and "String Variables" in this manual for more information about allocating memory for strings.

QUOTED STRINGS

Excessive use of quoted strings often occurs on larger computers because there is so much free memory. Shortening quoted strings may save memory. Also see ZBasic PSTR\$ function for an extremely efficient way of utilizing string constants in DATA statements and in regular statements.

EFFICIENT CODE

Careful examination of source code may uncover ways to decrease code size by making repeated commands into subroutines or FN's, or just cleaning up inefficiencies.

RAM DISKS

Some smaller computers allow the use of RAM disks. The Apple // ProDOS version for example, allows RAM disks up to 8 megabytes, while program and variable size are limited to 40K-50K. Utilizing a RAM disk to store indices, large arrays or whatever is nearly as fast as having that data in direct memory.

USE DISK INSTEAD OF MEMORY

If very large arrays or indices have been used in a large program you may have to store and access them from disk in a random file. This is slower than RAM access but is usually quite acceptable on most systems.

TEXT WIDTHS

Some computers have only 64 or 40 characters across the screen or 16 rows down the screen. You may have to adjust the program to accommodate this.

You should think about using the PRINT% or INPUT% commands if you plan on porting programs often. PRINT% puts text at ZBasics' device independent graphic coordinates, not text column/row coordinates. This makes porting programs much simpler. Here are some typical character layouts:

COMPUTER	Columns (across)	Rows (down)
IBM PC and compatible	80 or 40	25
Apple // series	80 or 40	24
TRS-80 Model I, III	64 or 32	16
TRS-80 Model 4, 4p	80, 64 or 32	16 or 24
CP/M (typical)	80	24
Macintosh	Almost anything...	See appendix

PORTING PROGRAMS

CHARACTER SETS

Screen and printer characters vary from one computer to the next. Check the ASCII chart in the owners manuals to see the differences. (Most between 32-127 are the same.)

KEYBOARD, JOYSTICK AND MOUSE

Keyboards vary from computer to computer so be sure the target computer has the same keys available. If not, make changes in the program to use other keys.

Joystick and MOUSE devices vary considerably. Test the controls on the target computer and make adjustments for the hardware.

SOUND

Sound tone may vary from machine to machine. Check program and make any adjustments needed. Some machines may not have this capability at all.

DEVICE INDEPENDENT GRAPHICS

ZBasic makes use of very powerful and simple graphic commands that work the same way regardless of the graphic capabilities of the target computer (or lack of).

You will have to determine if the graphic hardware on the target computer is of sufficient quality to display the graphics of your program. Note: Colors and grey levels may have to be adjusted. Here are some of the typical graphic types available for some major computers:

<u>COMPUTER</u>	<u>Horizontal x Vertical pixels</u>
IBM PC and compatibles	CGA: 640x200 (3 color) or 320x200 (8 color) EGA: 640x348 (many colors) HERCULES and HERCULES PLUS: 720x348 MDA: 80x25 (text simulation)
Apple //e, //c, //GS	Hi-Res 280x192 (6 color)
Apple //e, //c, //GS	Double Hi-Res 560x192 (16 color)
Macintosh	512,340 (larger monitors also supported)
TRS-80 Model I, III	128x48
TRS-80 Model 4,4p	160x72
	RS and Micro-Lab's hi-res boards 640-240
CP/M-80 (typical)	80x24 (text simulation)
KAYPRO with graphics	160x100

MACHINE DEPENDENT SUBROUTINES

If the program being ported contains machine language subroutines, you will need to rewrite those routines in the machine language of the target computer. Watch out for:

DEFUSR	USR	OUT	INP
MACHLG	LINE	CALL	
PEEK	PEEKWORD	PEEKLONG	
POKE	POKEWORD	POKELONG	

Unless you completely understand the machine language of both the target and source computer, use extreme caution when porting programs with these commands.

PORTING PROGRAMS

MACHINE SPECIFIC COMMANDS

In order to take advantage of unique or special features of some computers, ZBasic offers special commands that will not work or function on others. Be sure the program you are porting contains only commands from the reference section of this manual.

Special ZBasic commands may have to be rewritten for the target computer.

Be sure to read the ZBasic appendices for both the Target and Source computers. They will explain in detail the special commands for each system (you must purchase a version of ZBasic for each computer you wish to compile from).

METHODS OF TRANSFERRING SOURCE CODE FROM ONE MACHINE TO ANOTHER

Telephone Modem Transfer

Transfer files using a Modem and simple communications software routines like the ones under OPEN"C" in the main reference section of this manual.

Serial (RS-232) Transfer

Transfer files over the Serial (RS-232) ports of the two computers using a good communication software package like Crosstalk or SmartCom. Crosstalk is available at computer or software stores nationally.

Diskette File Transfer Utility Programs

Use Diskette file transfer utility programs like Uniform or Interchange. These programs will convert a file from one disk format, like from a TRS-80 diskette to another disk format, like MS-DOS or CP/M. These programs are available from computer or software dealers nationally.

Re-type the Program

Type the program into the other computers. This may be acceptable for small programs but you will save plenty of time by using one of the options above.

See OPEN"C" in the reference section for a ZBasic terminal routine that may be used to transfer files.



Important Note: Always transfer files in ASCII. Tokens are not necessarily the same from one version of ZBasic to another and from old versions to newer versions on the same machine.

CONVERTING OLD PROGRAMS



CONVERTING OLD PROGRAMS



CONVERTING PROGRAMS WRITTEN IN OTHER VERSIONS OF BASIC

ZBasic is a very powerful and improved version of BASIC. Many of the traditional BASIC commands have been retained to make conversion as easy as possible. Nevertheless, ZBasic is not 100% compatible with every BASIC. You will have to make some changes to your old programs if you wish to convert them to ZBasic.

If file and graphic handling are not used, conversion will normally be very simple. If files or graphics are used the conversion will take a little more thinking. The following pages will give you important insights into making the conversion process as easy as possible.

The following pages will give you some ideas about converting your older BASIC programs. Following the paragraphs step-by-step will make conversion much easier.

SAVE YOUR OLD BASIC PROGRAM AS ASCII OR TEXT

Save your old BASIC program in ASCII or TEXT format so it can be loaded into ZBasic. ZBasic tokens are different from other BASIC tokens so loading them without first converting them to ASCII will make programs loaded look like random control codes or the wrong commands (if the program will load at all).

See the owners manual for the older BASIC to determine how to save in ASCII or TEXT format for your computer. The typical syntax is; SAVE "filename",A.



Note: When upgrading to newer versions of ZBasic, programs may have to be saved in ASCII in the older version before loading into the newer version since tokens may have changed.

CONVERTING OLD PROGRAMS

CONFIGURING ZBASIC TO MAKE CONVERSIONS A LOT EASIER

ZBasic has been configured to give you maximum performance. When converting older BASIC programs this can be a problem. Often they are configured for ease of use instead of performance. ZBasic allows you to configure options so that converting your programs is simpler. Setting some of the options below will also make ZBasic more like BASIC you may be used to (like MSBASIC and BASICA).

Be sure to see "Configure" in the main reference section and in your appendix for details about other ways of configuring ZBasic.

To solve many of the problems encountered in converting we suggest setting the following options when converting other programs. Be sure to set these options BEFORE LOADING your program:

CONFIGURE OPTION	SET TO
1. Double precision digits of accuracy	6 or 8
2. Single precision Accuracy	4 or 6
3. Array bounds checking Y/N	Y
4. Default Variable type <S>ingle, <D>ouble, <I>nteger	S
5. Convert to Uppercase Y/N	Y
*6. Optimize expressions for Integer Y/N	N
*7. Spaces Required between Keywords Y/N	Y

1. Since ZBasic does all floating point operations in double precision, it is important to configure ZBasic for the speed and accuracy that you need. In most cases the configuration above will be suitable (but not in all cases). If you wish disk files and memory requirements to be the same as MSBASIC leave the digits of accuracy at 14 and 6 as they take up 8 bytes of Double and 4 bytes for single (the same as MSBASIC).

2. Set to two digits less than Double precision.

3. Sets array bounds checking to give runtime errors. Set to "N" when your program is debugged.

4. Set to Single (S) if you want code to be most like other BASICs. We highly recommend you set it to integer if possible. Integer will often increase program speeds 10 to 100 times.

5. Setting allows variables like "Fred" and "FRED" to be the same variable. If you want CASE to be significant, do not change the configuration.

6. ZBasic gives you two options for deciding how expressions may be evaluated. ZBasic defaults to optimizing expressions for Integer to get the fastest and smallest code. Most other languages do not. Set to "N" for easier conversions. See "Math" for explanation of ZBasic options for expression evaluations.

7. Some BASICs allow using keywords in variables (like INTEREST). To allow this, spaces or other non-variable type characters are required around keywords. Set this for easier conversion in most cases (especially IBM PC and Macintosh BASIC type programs).

*Note: Not available on all versions of ZBasic.

CONVERTING OLD PROGRAMS

CONVERTING RANDOM FILES

ZBasic incorporates FIELD, LSET, MKI\$, MKS\$, MKD\$, CVI, CVS and CVD into the READ and WRITE statements saving the programmer a lot of time. RECORD is used instead of GET and PUT for positioning the file pointer.

The OPEN and CLOSE statements are the same for both BASICs except for MSBASIC use of OPEN FOR RANDOM type. This is changed easily.

<u>ZBASIC statements</u>	<u>MSBASIC equivalents</u>
OPEN"R"	OPEN"R" or OPEN FOR RANDOM
READ, WRITE, RECORD	FIELD, GET, PUT, LSET, RSET, CVS, CVD, MKS\$, MKD\$, CVI, MKI\$

Note: While ZBasic also supports MKI\$, CVI and MKB\$, CVB, they are not necessary for use n Random files since ZBasic's READ and WRITE automatically store and retrieve numeric data in the most compact format (ZBasic's MKI\$, CVI, MKB\$ and CVB are most useful for condensing numbers for other reasons). Since ZBasic allows using any variable type in READ and WRITE statements, the user is not faced with complicated conversions of strings-to-numbers and numbers-to-strings.

CONVERTING SEQUENTIAL FILES

Most ZBasic Sequential file commands are very similar or the same to MSBASIC.

<u>ZBASIC statements</u>	<u>MSBASIC equivalents</u>
OPEN"I" or OPEN"O"	OPEN"I", OPEN"O" or OPEN"A" or OPEN FOR INPUT,
OPEN"A" some versions	OUTPUT or APPEND some versions
EOF(n) some versions	EOF(n) some versions
LINEINPUT, INPUT, PRINT	LINEINPUT, INPUT, PRINT

Note: The biggest difference when converting sequential file statements is that ZBasic's PRINT# statements should have quoted commas:

MSBASIC: PRINT#1, A\$,B\$,C\$ or PRINT#1, A\$ B\$ C\$
ZBASIC: PRINT#1, A\$" , "B\$" , "C\$

DISK ERROR TRAPPING

ZBASIC statement	MSBASIC equivalent
ON ERROR GOSUB	ON ERROR GOSUB

Read "ON ERROR" and "Disk Error Trapping" in this manual for detailed information. ZBasic error codes are much different from MSBASIC.



Important Note: ZBasic does not necessarily store data in disk files in the same way or format as other versions of BASIC. You may have to convert existing BASIC files to ZBasic format.

CONVERTING OLD PROGRAMS

CONVERTING GRAPHIC COMMANDS

ZBasic's Device Independent Graphics are very powerful and simple to understand. Conversion should be painless in most cases:

ZBASIC GRAPHICS

PLOT
CIRCLE
BOX
COLOR
MODE
POINT
GET, PUT (some systems)
RATIO
FILL
PLOT USING

MSBASIC equivalent

LINE,PSET,PRESET
CIRCLE
LINE (with parameters)
COLOR (PSET, PRESET black and white)
SCREEN
POINT
GET, PUT (some systems)
aspect parameter of CIRCLE
PAINT
DRAW

ZBasic defaults to a relative coordinate system of 1024x768. This system does not pertain to pixels but to imaginary positions on the screen. Most older versions of BASIC use pixel coordinates.



Macintosh and MSDOS: Use COORDINATE WINDOW at the beginning of program to set a program to pixel coordinates. Apple: See appendix for ways of using POKE to set system to pixel coordinates.

LOOP PAIRS

All ZBasic FOR-NEXT, WHILE-WEND and DO-UNTIL loops must have matching pairs. Some BASIC interpreters allow the program to have two NEXTs for one FOR, or two WENDs for one WHILE. Since ZBasic is a compiler it will not allow this. A STRUCTURE ERROR will be generated when you compile a program with unmatched LOOP pairs.

Another way to find unmatched pairs is to LIST a program. Since ZBasic automatically indents loops, just read back from the end of the LISTing, looking for the extra indent, to find the unmatched statement.

COMPLEX STRINGS

Complex strings may have to be converted to simple strings (some machines).

Improper B\$=LEFT\$(Right\$(A\$,12), 13)
Proper B\$=RIGHT\$(A\$,12): B\$=LEFT\$(B\$,13)

IF-THEN statements may have only one level of complex string.

Improper IF B\$=LEFT\$(A\$,5) THEN GOSUB "END"
Proper C\$=LEFT\$(A\$,5): IF B\$=C\$ THEN GOSUB "END"

CONVERTING OLD PROGRAMS

LONG LINES

Multiple statement lines with over 253-256 characters (depending on computer) will automatically be shortened by ZBasic when loading. That part of the line longer than 253 will be added to a new line number. Most programs do not have lines of that length.

TIMING LOOPS

Timing loops may have to be lengthened to make up for ZBasic's faster execution time. For some BASIC Languages a FOR-NEXT loop of 1000 would take second or two. (About 1/1000 of a second in ZBasic!) Replace these types of delay loops with the ZBasic DELAY statement.

STRING MEMORY ALLOCATION



Important Note: ZBasic assumes a 255 character length for every string and string array element and allocates 256 bytes for each (255+1 for length byte) unless string length is defined with DIM or DEF LEN.

Many versions of BASIC, like BASIC^{Atm}, MSBASICtm, APPLESOFTtm and others, allocate string memory as a program needs.

While this may seem efficient on the surface, immense amounts of time are wasted in "String Garbage Collection". Garbage Collection is what happens when your program suddenly stops and hangs up for two or three minutes while BASIC rearranges strings in memory. This makes this method unusable for most serious programming.

HOW DIMMING STRING ARRAYS AFFECT PROGRAM CONVERSION

MSBASIC™:	CLEAR 10000 DIM A\$(1000)	Sets aside 10,000 bytes for ALL strings Uses memory allocated with CLEAR plus 3-8 byte pointers per element.
ZBASIC™:	DIM A\$(1000)	256,256 bytes allocated (100x256)
ZBASIC™:	DIM 10 A\$(1000)	10,010 bytes allocated (1001x10)

Many BASICs use CLEAR to set aside memory for strings. Each string in ZBasic is allocated memory at compile time.

A problem you may encounter while converting: Out of Memory Error from DIMension statements, like the ones above (just define the length of the string elements).

ZBasic allows you to define the length of any string with DEFLEN or DIM statements. Check the string requirements of the program you wish to convert and set the lengths accordingly.

If you have large string arrays that must have elements with wide ranging lengths (constantly changing from zero to 255 characters), use ZBasic's special INDEX\$ string array. Like other BASIC's CLEAR is used to set aside memory for this array (no "Garbage collecting" here either).

See INDEX\$, DEFLEN, DIM and "String Variables" for more information.

CONVERTING OLD PROGRAMS

OTHER INFORMATION

Check your appendix for more information about converting programs.

A good resource for information about converting from one version of BASIC to another is David Lien's "The BASIC Handbook".

CONVERTING OLD COMMANDS

Some BASIC(s) have commands that may be converted over quickly using a word processing program. Simply load the BASIC ASCII file into the word processor and use the FIND and REPLACE commands. (You may also use ZBasic FIND command if you choose.)

A good example would be converting Applesoftm's HOME commands into ZBasic's CLS command. Have the word processor FIND all occurrences of HOME and change them to CLS.

If you don't have a word processor try using this simple ZBasic convert program to change commands in a BASIC file quickly (file MUST have been saved in ASCII using SAVE*).

SINGLE COMMAND CONVERSION PROGRAM

```
ON ERROR GOSUB "DISK ERROR": REM Trap Disk Error
INPUT"Command to Change:";Old$
INPUT$"Change to:";New$
CLS: PRINT" Changing File...One Minute please"
OLDFILE$="oldfile":NEWFILE$="newfile": REM <-- Change to correct filenames
OPEN"I",1, OLDFILE$
OPEN"O",2, NEWFILE$
WHILE ERROR=0
  LINEINPUT#1, Line$
  DO
    Line$=LEFT$(Line$,I-1)+New$+RIGHT$(Line$,LEN(Line$)-I+1+LEN(Old$))
    I=INSTR(1, Line$, Old$)
  UNTIL I=0
  PRINT#2, Line$
WEND
"Done changing"
ERROR=0
CLOSE
PRINT "All `";Old$;`" have been converted to `";New$;`""
INPUT"Rename OLD file? Y/N: ";A$: A$=UCASE$(A$)
IF A$="Y" THEN KILL OLDFILE$
RENAME "NEWFILE" TO OLDFILE$
END
"DISK ERROR"
PRINT ERRMSG$(ERROR)
CLOSE: STOP
```

Important: Practice on a dummy file until you are sure the program is working properly.

KEYWORDS



STANDARD STATEMENTS, FUNCTIONS AND OPERATORS

ABS	FIX	MOD	SIN
AND	FN	MODE	SOUND
ASC	FOR	MOUSE	SPACE\$
ATN	FRAC	NEXT	SPC
BIN\$	GOSUB	NOT	SQR
BOX	GOTO	OCT\$	STEP
CALL	HEX\$	ON	STOP
CHR\$	IF	OPEN	STR\$
CIRCLE	INDEX\$	OR	STRING\$
CLEAR	INDEXF	OUT	SWAP
CLOSE	INKEY\$	PAGE	TAB
CLS	INP	PEEK	TAN
COLOR	INPUT	PLOT	THEN
COS	INSTR	POINT	TIMES\$
CVB	INT	POKE	TO
CVI	KILL	POS	TROFF
DATA	LEFT\$	PRINT	TRON
DATE\$	LEN	PSTR\$	UCASE\$
DEF	LET	RANDOM	UNS\$
DEFDBL	LINE	RATIO	UNTIL
DEFINT	LOC	READ	USING
DEFSNG	LOCATE	REC	USR
DEFSTR	LOF	RECORD	VAL
DELAY	LOG	REM	VARPTR
DIM	LONG	RENAME	WEND
DO	LPRINT	RESTORE	WHILE
ELSE	MACHLG	RETURN	WIDTH
END	MAYBE	RIGHT\$	WORD
ERRMSG\$	MEM	RND	WRITE
ERROR	MID\$	ROUTE	XELSE
EXP	MKB\$	RUN	XOR
FILL	MKI\$	SGN	



IMPORTANT: See your computer appendix for other keywords that pertain to your version of ZBasic. Most versions of ZBasic offer more and also use two-word keywords like LONG FN, POKE WORD etc.

KEYWORDS

STANDARD COMMANDS

APPEND	HELP	NEW
AUTO	LIST, L or period "."	QUIT
DELETE or DEL	LLIST	RENUM
DIR	LOAD	RUN
EDIT, E or comma ","	MEM	SAVE
FIND or semicolon ";"	MERGE	



STANDARD REFERENCE GLOSSARY

This reference section is an alphabetical listing of the "Standard ZBasic Commands". The following paragraphs describe the information layout and syntax of this section.

TYPE OF INFORMATION CONTAINED IN THIS REFERENCE SECTION

function	Returns a value; used wherever an expression is used
statement	Executed by itself
command	Used from the standard line editor mode; EDIT, SAVE...
operator	Like AND, OR, XOR or NOT

COMPATIBLE COMMANDS

 BLACK BAR Indicates the command is the same on all versions of ZBasic.

 SPECKLED BAR Indicates the command may not be available on all versions. Check to see if your system does not support that command.

PAGE LAYOUT

The pages are layed out in the same way. Whenever possible descriptions are kept to one page. The header has the command type and description. Paragraph layout is:

FORMAT	Correct syntax for that statement, function or command
DEFINITION	Definition or explanation of usage
EXAMPLE	Program example or direct example of usage. Note that linenumbers are usually omitted. Add linenumbers if needed.
REMARK	Other information of importance and usually a reference to other related sections that will aid the understanding of that item.

IMPORTANT NOTE ABOUT DIVIDE

ZBasic compiles divide symbols based on configuration.

If the default expression evaluator; "Optimize Expressions as Integer?" is YES;

/=integer divide \=floating point divide

If the expression evaluator; "Optimize Expressions as Integer?" is NO;

/=floating point divide \=integer divide

See "Configure" and "Converting Old Programs" and "Math expressions" for more information about the options offered for expression types and how they are evaluated.

continued next page...

STANDARD REFERENCE

CROSS REFERENCE

These commands work the same way on almost every version of ZBasic. There is an extensive cross-reference to other commands and how a command works on specific machines. The reference section uses a computer icon to bring attention to a specific version of ZBasic. The following icons are used:



Apple // DOS 3.3 and ProDOS versions.



MSDOS and IBM PC and compatible versions.



The Macintosh versions (all except the 128k machine).



Z80 machines; Amstrad, CP/M-80 2.x and higher, Kaypro Graphics versions and TRS-80 model 1, 3 and 4 versions.

SYNTAX GLOSSARY

GLOSSARY

RUN or COMMAND

[brackets]

{ A|B|C }

... repeats

Courier text

expression or expr

byte expression

word expression

long expression

variable or var

var\$, var%, var&, var!, var#

"string"

simplestring or string

filename

filename

filespec

line

number

var name

DEFINITION

What follows is program or command output.

Items within the brackets are optional (may be omitted)

Any one of A, B or C may be used

Three periods following items indicates a repeating sequence

Something you type in, a program example, or program output

Numeric: Any; including integer and floating point

Numeric: 0-255

Numeric: 0 to 65,535 or +-32,767

Numeric: 0 to 4,294,966,293 or +-2,147,483,647

Any Variable

String, integer, LongInteger, single or double precision variable types, respectively

Quoted strings (string constants)

String variable, string constant, BIN\$, CHR\$, HEX\$, INDEX\$, OCT\$, PSTR\$, STR\$, SPACE\$, STRING\$ or UNS\$.

File number: An expression 1-99. See "Configure"

A legal filename for that operating system filename

Drive or storage volume specifier

A line number from 0 to 65,534 or a "label"

Requires a number. No variable or expression allowed

A valid variable name



Be sure to take note when you see this hand. It is pointing out important information about using that command. If there is the message "Important Note" with the hand it is even more critical that you read the notes.

FORMAT **ABS** (expression)

DEFINITION Returns the absolute value of an expression. The absolute value is the value without regard to the sign (negative, zero or positive).

The result of ABS will always be a positive number or zero.

EXAMPLE A=-15: B=15
 PRINT ABS(A), ABS(B), ABS(-555)
 X=ABS(0)
 PRINT X

RUN

15, 15, 555
0

REMARK The SGN function will return the sign of an expression.

AND operator

FORMAT expression1 **AND** expression2

DEFINITION Used to determine if BOTH conditions are true. If both expression1 AND expression2 are true (non-zero), the result is true. Returns -1 for true, 0 for false.

Also used to compare bits in binary number operations. 1 AND 1 return a 1, all other combinations of 0's and 1's produce 0. See truth tables below.

EXAMPLE IF 30>20 AND 20<30 THEN PRINT "TRUE "
IF "Hi"="hello" AND 6-5=1 THEN PRINT "TRUE TOO!"

RUN

TRUE

PRINT BIN\$(&X00001111 AND &X11111111)
PRINT 4 AND 255

RUN

0000000000001111
4

REMARK See OR, XOR and NOT.

AND TRUTH TABLE

condition AND condition TRUE(-1) if both conditions TRUE, else FALSE(0)

AND	BOOLEAN "16 BIT" LOGIC	
1 AND 1 = 1	00000001	00000111
0 AND 1 = 0	AND 00001111	AND 00001111
1 AND 0 = 0	= 00000001	= 00000111



LongInteger will function with this operator in 32 bits.

command APPEND

FORMAT **APPEND** line or label ["] filename["]
APPEND* line or label ["] filename["]

DEFINITION Used to append or insert a program segment or subroutine (saved with SAVE+) into the present program in memory.

A non-line numbered ASCII program file is required to append a subroutine into the present program in memory at the specified line number. Line numbers will be assigned in increments of one.

APPEND* will strip REM(arks) and spaces to free up more memory for the program as the program is inserted.

EXAMPLE

```
10 "TEST ROUTINE"  
20 FOR I = 1 TO 10  
30 PRINT I  
40 NEXT I  
50 RETURN
```

```
SAVE+ TEST.APP
```

```
APPEND 31 TEST.APP
```

LIST

```
00010 "TEST ROUTINE"  
00020 FOR I = 1 TO 10  
00030 PRINT I  
00031 "TEST ROUTINE" <----Subroutine inserted here  
00032 FOR I = 1 TO 10 <----(Example only, program will not run)  
00033 PRINT I  
00034 NEXT I  
00035 RETURN  
00040 NEXT I  
00050 RETURN
```

REMARK

The program to be appended must be in ASCII format and not contain line numbers. Use the SAVE+ command to save programs without line numbers.

If any line number being used in APPEND already exists, it will overwrite the existing line. Also see MERGE, LOAD, SAVE, SAVE*, SAVE+.

ASC function

FORMAT **ASC**(string)

DEFINITION Returns the ASCII code value (a number between 0 and 255) of the first character in a string. ASCII stands for American Standard Code for Information Interchange.

EXAMPLE

```
PRINT ASC("A"), ASC("B")
PRINT CHR$(65), CHR$(66)
PRINT ASC("America")
```

RUN

```
65                    66
A                     B
65
```

REMARK ASC returns 0 if the length of string is zero or the ASCII code of the string is zero. Use this logic to determine the true status if an ASCII zero is the result:

```
LONG IF ASC(A$)=0 AND LEN(A$)>0
  PRINT "ASCII code of A$ =0"
XELSE
  PRINT"A$ is an empty string"
END IF
```

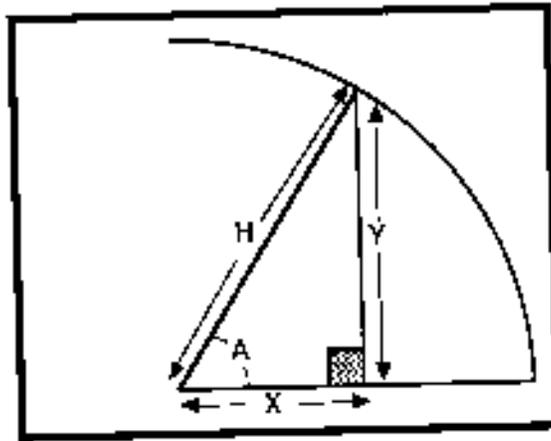
The inverse function of ASC is CHR\$. To return the character represented by the ASCII code, use CHR\$(ASCII number)

ASCII codes may vary from machine to machine.

ASCII codes 32 through 127 are usually the same for all microcomputers. See CHR\$ with example ASCII listing.

FORMAT ATN(expression)

DEFINITION Returns the angle, in radians, for the inverse tangent of expression.



$A = \text{ATN}(Y/X)$, $P = \text{ATN}(1) \ll 2$

EXAMPLE

```
Pi#=ATN(1) << 2
PRINT Pi#
```

RUN

3.141592... <---Based on digits of accuracy set in configuration.

REMARK ATN is a scientific function. Using ATN in an expression will force ZBasic to calculate that part of an expression in Double Precision.

ZBasic allows you to configure the accuracy for scientific functions separately for both Double and Single Precision. See "Configure".

Also see "Expressions" and "Derived math functions" in the "MATH" section of this manual.

statement BEEP

FORMAT **BEEP**

DEFINITION Sounds the speaker.

EXAMPLE

```
FOR X=1 TO 10
  BEEP
NEXT

RUN

BEEP, BEEP...
```

REMARK Also see SOUND.



BEEP is not supported with Apple // or Z80 computers. For Apple // and most CP/M computers use `PRINT CHR$(7)` instead. See your SOUND and your computer appendix for other ways of creating audio output.

BASE OPTION configuration

FORMAT **Array Base** **0 or 1?**

DEFINITION An option in the ZBasic configuration routine to set the array BASE to either zero or 1. The default is zero.

EXAMPLE See "Configure" in the beginning of this manual for an explanation of configuring your version of ZBasic to your preferences.

ARRAY BASE ZERO

DIM A(100) <-- elements 0-100 (101 elements)
DIM Tables(22) <-- elements 0-22 (23 elements)

ARRAY BASE ONE

DIM A(100) <-- elements 1-100 (100 elements)
DIM Tables (22) <-- elements 1-22 (22 elements)

REMARKS See DIM and "Array Variables".

FORMAT BIN\$ (expression)

DEFINITION Returns a 16 character string which represents the binary (BASE 2) value of the result of the integer expression. Some typical binary numbers:

0000000000000001	=	1
0000000000000011	=	3
0000000000000111	=	7
0000000011111111	=	255
0000000100000000	=	256
1111111111111111	=	-1 (65,535 unsigned)

EXAMPLE The following program will convert a decimal number to binary or a binary number to decimal:

```
"Binary Conversion"
CLS
DO
  INPUT"Decimal number to convert: ";Decimal%
  PRINT BIN$(Decimal%)
  INPUT"Binary number to convert: ";Binary$
  Binary$="&X"+Binary$
  PRINT VAL(Binary$)
UNTIL Decimal% = 0
```

RUN

```
Decimal number to convert: 255
0000000011111111

Binary number to convert: 0000000000000011
3
```

REMARK Note that conversions are possible from any base to any other base that ZBasic supports. &X is the inverse function of BIN\$.

Also see HEX\$, OCT\$, UNS\$ and "Numeric Conversions".



Use DEFSTR LONG to set BIN\$ and &X to work in LongInteger (32bits).

BOX statement



FORMAT **BOX [TO] expr x1, expr y1 [TO expr x2,expr y2 ...]**
BOX FILL [TO] expr x1, expr y1 [TO expr x2,expr y2 ...]

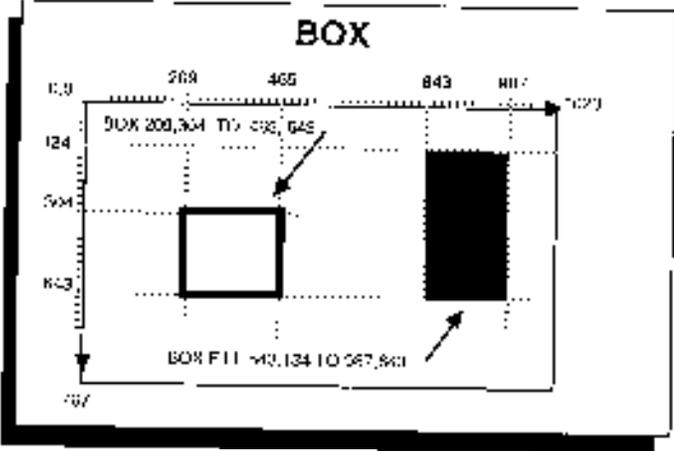
DEFINITION Draws a BOX from the coordinates defined by the first corner (x1,y1) to the coordinates defined by the opposite corner (x2,y2) in the current COLOR.

If BOX TO x,y is used the first corner will be the last graphic point used. If undefined then 0,0 will be the default.

If the optional FILL appears directly after the command, the BOX will be painted as a solid BOX in the current color.

The default screen positions are given using Device Independent Coordinates of 1024 across by 768 down.

EXAMPLE



REMARK The output will vary depending on the graphic capability of the host computer. Also see CIRCLE, MODE, FILL, PLOT, RATIO and COLOR.

FORMAT **CALL** number
 CALL LINE line or label

DEFINITION **CALL** will execute a machine language subroutine at the address specified by number or the address of the compiled line.

EXAMPLE Use these examples only if you understand machine language.

```

REM TRS80 I & III, CALL DEBUG
CALL &H440D
:
REM CPM 80, CALL WARM START (Exits to DOS)
CALL 0
:
REM APPLE CALL TO SOUND BELL TONE
CALL -198
:
10 REM CALL LINE examples
20 CALL LINE 40
30 CALL LINE "LABEL"
40 MACHLG 34, 21, x%, 255, 9: RETURN
50 "LABEL": MACHLG . . . : RETURN
  
```

REMARK **CALL** is useful for transferring program control to a machine language subroutine from which a return to the ZBasic program is desired. The routine to be called must be terminated by that machine's instruction for RETURN.

Also see MACHLG, USR, LINE and DEFUSR.



WARNING: Use of this command requires an understanding of machine language programming and the computer hardware being used. Porting of this code may not be possible without re-writing the machine language routines.



See CALL in your appendix for enhancements.

CASE statement

FORMAT	<pre>SELECT [CASE] [expression] CASE [IS] relational condition [, relational condition] [...] statement [:statement:...]] CASE [IS] condition [, condition] [...] statement [:statement:...]] CASE boolean expression statement [:statement:...]] CASE ELSE statement [:statement:...]] END SELECT</pre>
DEFINITION	<p>When SELECT/CASE is encountered, the program checks the value of the controlling expression or variable, finds the CASE that compares true and executes the statements directly following the CASE statement. After these statements are performed, the program continues at the line after the END SELECT statement:</p>
CASE relational,...	<p>If the expression after SELECT compares true to any one of a number of relational conditions, the statements following the CASE are executed and the program continues after the END SELECT:</p> <pre>SELECT 12 CASE >10 PRINT "This is the right answer" CASE >20, <10 PRINT "This is not true" END SELECT</pre> <p>program continues here...</p>
CASE condition,...	<p>If the expression following SELECT equals any one of a number of conditions the statements following the CASE are executed (program continues after the END SELECT).</p> <pre>A=23 SELECT A CASE 10 PRINT "This is the wrong answer" CASE 10,23,11,10 PRINT "This would be true" END SELECT</pre>
CASE boolean	<p>If an expression after SELECT is omitted, you may use a boolean or TRUE/FALSE condition. The statements after the first TRUE (non-zero) CASE condition will be executed. Only one boolean statement is allowed following CASE.</p> <pre>A=10:B=20 SELECT CASE (A=10 AND A>20) PRINT "This is the correct answer" CASE (A>B OR A=B) PRINT "This is the wrong answer" END SELECT</pre>

statement CASE

CASE ELSE

If all of the CASE statements in the SELECT CASE structure are false the statements following the CASE ELSE are executed.

```
"Start"  
A$="Maybe"  
SELECT A$  
CASE "Yes"  
    PRINT "Thank you for saying Yes"  
CASE "No"  
    PRINT "Thank you for saying No"  
CASE ELSE  
    PRINT "You smart aleck!"<--Does this one  
END SELECT
```

REMARK

This is a powerful structured way of doing complicated IF-THEN-ELSE or LONG IF statements especially when there are multiple lines of complicated comparisons.

This structure is also much easier to read than complicated IF statements.

See SELECT for more information.



Important Note: Never exit a SELECT CASE structure using GOTO. This will introduce problems into the stack and cause unpredictable system errors. Always exit the structure at the END SELECT. Be sure to enclose loops and other constructs completely within the SELECT-CASE and CASE ELSE constructs.



The Z80 versions do not support SELECT CASE. See LONG IF and IF for ways of doing the same thing.



The Apple DOS 3.3 and ProDOS versions does not support SELECT CASE. See LONG IF and IF for ways of doing the same thing.

CHR\$ function

FORMAT **CHR\$** (expression)

DEFINITION Returns a single character string with the ASCII value of the result of expression. The range for the value of expression is 0 to 255.

The inverse function of CHR\$ is ASC;

EXAMPLE

```
"Print ASCII character set for this computer"
CLS
REM Use ROUTE 128 here to send output to printer.
FOR I=32 TO 127 STEP 8
  FOR J= 0 TO 7: X =I+J
    PRINT USING "###=";X;CHR$(X); " ";
  NEXT J  :PRINT
NEXT I

RUN

32=  33=!  34="  35=#  36=$  37=%  38=&  39='
40=(  41=)  42=*  43=+  44=,  45=-  46=.  47=/
48=0  49=1  50=2  51=3  52=4  53=5  54=6  55=7
56=8  57=9  58=:  59=;  60=<  61==  62=>  63=?
64=@  65=A  66=B  67=C  68=D  69=E  70=F  71=G
72=H  73=I  74=J  75=K  76=L  77=M  78=N  79=O
80=P  81=Q  82=R  83=S  84=T  85=U  86=V  87=W
88=X  89=Y  90=Z  91=[  92=\  93=]  94=^  95=_
96=`  97=a  98=b  99=c  100=d  101=e  102=f  103=g
104=h  105=i  106=j  107=k  108=l  109=m  110=n  111=o
112=p  113=q  114=r  115=s  116=t  117=u  118=v  119=w
120=x  121=y  122=z  123={  124=|  125=}  126=~  127=#
```

```
PRINT CHR$(64)
PRINT ASC("A")

RUN

A
64
```

REMARK

When the program above is run, the character set for that computer will be displayed. Some of the characters above may differ from what you get on your system. Try changing the range above from 127 to 255. Some computers have extra characters or graphic symbols for these codes.

Characters in the range of 0-31 are usually reserved for control codes like linefeed (10), carriage return (13)...

If the PRINT statement is changed to LPRINT the printer's character set will be printed. If expression is less than 0 or greater than 255, only the low order byte will be used.

```
CHR$(256) = CHR$(0)
CHR$(257) = CHR$(1)
```

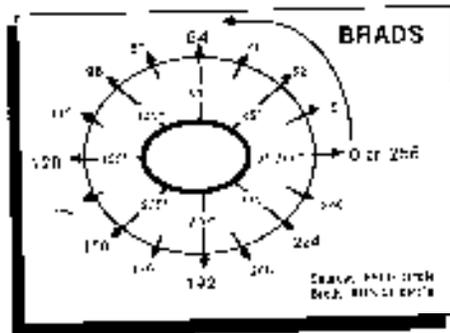
statement CIRCLE

FORMAT **CIRCLE [FILL]** *expr1, expr2, exprR*
CIRCLE *expr1, expr2, exprR* **TO** *exprS, exprB*
CIRCLE *expr1, expr2, exprR* **PLOT** *exprS, exprB*

DEFINITION Draws a CIRCLE in the current COLOR.

If the optional FILL is used directly after the command, the CIRCLE will be filled with the current COLOR. If TO is used, a PIE segment will be displayed (shaped like pie slices). If PLOT is used, only the ARC segment will be displayed (a segment of the circumference).

expr1 horizontal center
expr2 vertical center
exprR radius (diameter of circle) in graphic coordinates
exprS start of angle in brads (zero starts at 3:00 o'clock)
exprB Number of brads to draw ARC or PIE (counter clockwise).



EXAMPLE SEE ILLUSTRATIONS OF FOLLOWING PAGE.

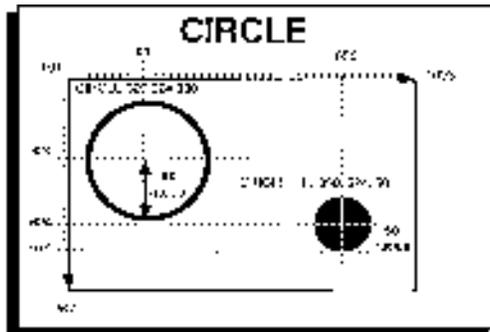
REMARK CIRCLE uses the ZBasic Device Independent Graphic Coordinates of 1024 x 768. For more details see the CIRCLE in the "Graphics" section in this manual. Also see RATIO,MODE,PLOT,COLOR,FILL and BOX.



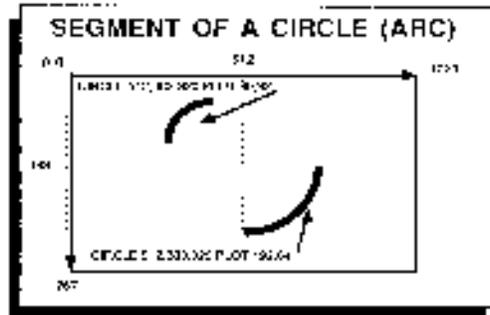
Macintosh: See COORDINATE WINDOW for pixel coordinates and toolbox for ways of using QuickDraw for creating boxes. MSDOS: See COORDINATE WINDOW for converting to pixel coordinates. Apple: See appendix for ways of converting to pixel graphics.

CIRCLE statement

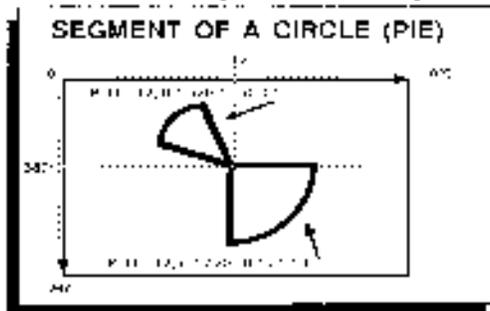
EXAMPLE CIRCLE *expr1*, *expr2*, *exprR*
 CIRCLE FILL *expr1*, *expr2*, *exprR*



CIRCLE *expr1*, *expr2*, *exprR* PLOT *expr2*, *exprR*



CIRCLE *expr1*, *expr2*, *exprR* TO *expr3*, *exprR*



statement CLEAR

FORMAT **CLEAR**
 CLEAR *number*
 CLEAR END
 CLEAR INDEX\$

DEFINITION Used to reserve memory or clear all or specified variables (sets the values of the variables to null or zero).

CLEAR Sets all variables and INDEX\$ to zero or null.

CLEAR number Sets aside number bytes for the INDEX\$ array.
CLEAR END CLEARS all variables which have not yet been assigned in the program. This form of CLEAR is normally used to clear all variables not being used when chaining. See "Chain" in the front section for more information.

CLEAR INDEX\$ Sets all elements of the INDEX\$ array to null.

EXAMPLE INPUT "Name: " ;Name\$
 PRINT Name\$
 CLEAR
 PRINT Name\$

RUN

Fred

<-----Nothing printed here since Name\$ was cleared at line 3.

REMARK Only one CLEAR number is allowed in a program and must appear before any variables are encountered. Be sure to CLEAR one extra byte for each element in the INDEX\$ array. Also see "Special INDEX\$ Array" and "CHAIN".

A CLEAR is performed at the beginning of each program created with RUN or RUN*. RUN+ or warm start programs will not CLEAR variables at startup.



See INDEX\$ in Mac appendix for added enhancements available on this version.

CLOSE statement

FORMAT **CLOSE** [[#] *expression*₁ [, [#] *expression*₂...]]

DEFINITION This statement is used to CLOSE one or more OPEN files or other devices.

The parameter expression indicates a device number or file number.

If no file or device numbers are declared all OPEN devices will be closed.

EXAMPLE

```
OPEN" I" ,1, "FILE1" ,10
OPEN" I" ,2, "FILE2" ,100
READ#1, A$;10
READ#2, B$;10
CLOSE#1,2
OPEN" R" ,1, "FILE3"
CLOSE
```

<---File1 and 2 are closed
<---File1 may now be used again
<---All files are closed

REMARK

All files should be closed before leaving a program to insure that data will not be lost or destroyed. If a program exit is through END or STOP, all files will be closed.

FORMAT **CLS**
CLS *expression*
CLS LINE
CLS PAGE

DEFINITION These statements will clear all, or portions, of the screen of text and graphics.

CLS	Clears the entire screen of text and graphics. Cursor ends up at the top left corner of screen.
CLS <i>expression</i>	In TEXT mode this fills screen with the ASCII character specified by <i>expression</i> and places the cursor at the top left corner of the screen*.
CLS <i>expression</i>	In GRAPHICS mode this will fill the screen with the color specified by <i>expression</i> .
CLS LINE	Clears from the cursor position to the end of the line. Cursor will remain where it was.
CLS PAGE	Clears from the cursor position to the end of the screen. Cursor will remain where it was.

EXAMPLE

CLS	
CLS 65	<----Fills screen with A's
CLS ASC(" * ")	<----Fills screen with *'s
LOCATE 0,10	
CLS LINE	<----Clears line 10 of text and graphics
LOCATE 0,12	
CLS PAGE	<----Clears screen from line 12 down.

REMARK See LOCATE,PRINT@,PRINT%,FILL and MODE. See your computer appendix for possible variations.



CLS clears the current window (not the entire screen). CLS *expression* will clear the screen with white if *expression*=0 and black if *expression*><0.

COLOR statement

FORMAT **COLOR** [=] *expression*

DEFINITION Sets the COLOR to be used by all graphic drawing commands. Color values will vary from one computer to the next. See your computer appendix for specifics. For most computers 0 is the background color and -1 is the foreground color.

If you have a black and white monitor, 0 is Black, -1 is white.

If your computer is incapable of graphics or your are using one of the character modes, the expression will determine the ASCII character to be used. (With some graphics modes, zero=space, all others=asterisk "*").

EXAMPLE

```
CLS: MODE 6                      <----even modes are character graphics with some versions
COLOR ASC( "*" )               <----Uses asterisks for graphics (not all versions)
PLOT 0, 256
MODE=7                         <----odd modes are actual graphics
CIRCLE 768,200,50
COLOR=6                       <----Sets COLOR to 6
BOX 0,0 TO 10,10
END
```

REMARK Also see MODE,PLOT,CIRCLE,BOX,POINT and FILL. Colors vary by mode, graphic type, monitors and other hardware criteria. Check hardware manual and the ZBasic appendix for your computer for specific color codes.



Macintosh: NOT(0) =black, 0=white. See appendix for variations especially with Macintosh II which supports a number of colors and grey levels.

MSDOS: COLOR is also used to change text color, background color, blinking, underline etc. See appendix for specifics. See CGA colors below.

Apple: Color chart below and the Apple appendix.

TRS-80 and Kaypro: Black=0, -1=white.

EXAMPLE COLORS CODES

IBM PC and compatibles CGA MODE 5		Apple // ProDOS and DOS 3.3 MODE 5 MODES 1,3 and 7		
0= BLACK	8=GRAY	0=BLACK1	0=BLACK	8=BROWN
1=BLUE	9=LT BLUE	1=GREEN	1=MAGENTA	9=ORANGE
2=GREEN	10=LT GREEN	2=VIOLET	2=DARK BLUE	10=GREY
3=CYAN	11=LT CYAN	3=WHITE1	3=PURPLE	11=PINK
4=RED	12=LT RED	4=BLACK2	4=DARK GREEN	12=GREEN
5=MAGENTA	13=LT MAGENTA	5=ORANGE	5=GREY	13=YELLOW
6=BROWN	14=YELLOW	6=BLUE	6=MED. BLUE	14=AQUA
7=WHITE	15=Bright WHITE	7=WHITE2	7=LIGHT BLUE	15=WHITE

statement COMMON

FORMAT COMMON variable list...

DEFINITION Identical to the ZBasic DIM statement. It is used to allocate memory for variables and for declaring variables common to chained programs.

The order of the variables declared in COMMON is important when chaining programs. The COMMON statement in one program must be exactly the same and in exactly the same order in other programs being chained.

EXAMPLE See DIM.

REMARK See DIM and "Chaining" in this manual.

This statement is added to make ZBasic compatible with other versions of BASIC.



Not available on the Apple // or Z80 version of ZBasic. Use DIM.

COMPILE command

FORMAT [L] **COMPILE**

DEFINITION Compiles a program and lists all of the compile time errors that are encountered.

If optional "L" is used, the error listings are sent to the printer.

This command is essentially the same as RUN except the compiler does not stop at the first error.

EXAMPLE

```
PWINT "Hello"  
X=X+1  
INPUT "Yes or No:"A$  
GOSUB "Routine"  
END
```

COMPILE

```
Syntax Error in Stmt 01 at Line 00001  
00001 PWINT "Hello"
```

```
;" Expected Error in Stmt 01 at line 00003  
00003 INPUT "Yes or No:"_A$
```

```
Line# Error in Stmt 01 at Line 00004  
00004 GOSUB "Routine"
```

REMARK See RUN and the section in the front of the manual called "Errors".



Not supported. Use RUN.



Not supported. Use RUN.

command CONFIG

FORMAT CONFIG

DEFINITION Invokes the configuration prompts that allow you to set preferences for a number of items including:

- Digits of precision
- Default variable types
- Integer or floating point expression evaluation
- Spaces between keywords
- Convert to uppercase
- Number of files that can be opened
- The Rounding factor for PRINT USING
- Test Array bounds

and a number of special options for your computer.

EXAMPLE See "Configure" in the front of this manual and the section in your appendix for specific configuration options available for your version of ZBasic.

REMARK This command is not available on all versions. See below.



The Z80 versions of ZBasic do not offer this command. The option to configure is offered only when you first load ZBasic.



CONFIG is not offered as a command but "Configure" is always available as a menu item. See appendix for the options specific to this version.

COORDINATE statement

FORMAT **COORDINATE** **[[WINDOW]** *horizontal, vertical*]

DEFINITION Allows you to change the coordinate system used for graphic functions and statements.

ZBasic defaults to a coordinate system of 1024 x 768. This allows programs created on one computer work on other computers with different graphic hardware.

COORDINATE *horiz,vert* Set the relative coordinate system to the specified limits minus one. **COORDINATE** 100,100 would allow setting the coordinates from 0 to 99 for both the horizontal and vertical.

COORDINATE WINDOW Sets the system to pixel coordinates. This allows you calculate the graphic positions by the actual resolution of the screen. While this is not recommended for programs that will be ported to other computers, some people prefer it for certain applications.

EXAMPLE

<code>PLOT 1023, 767</code>	<--- Puts a graphic dot at the ZBasic
<code>:</code>	default coordinates (lower right corner)
<code>COORDINATE WINDOW</code>	
<code>PLOT 100,100</code>	<--- Puts a graphic dot at the pixel coordinate
<code>:</code>	
<code>COORDINATE 1000,500</code>	
<code>PLOT 100,100</code>	<--- Puts a graphic dot at the relative coordinate

REMARK Some versions do not support this statement. See below for alternatives to changing coordinate systems.



Not supported on Z80 versions although **COORDINATE WINDOW** may be emulated by using this instruction: `POKE&xx3F,&C9` to enable pixel graphics and `POKE&xx3F,&C3` to return to the default coordinates of 1024x768. The value of xx varies by version type: CP/M-80=01, TRS-80 1,3=52 and TRS-80 model 4=30.



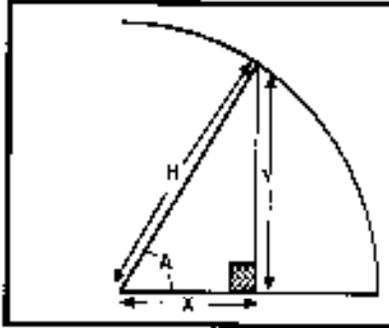
Not supported on these versions although **COORDINATE WINDOW** may be emulated using the statements below:

Apple ProDOS: `POKEWORD &85,0` for pixel coordinates for that mode of graphics. Use `MODE` to set back to regular coordinates.

Apple DOS 3.3: `POKE &F388,&60` for pixel coordinates of that mode. `POKE &F88,&A9` to set back to the default coordinates of 1024x768.

FORMAT **COS** (expression)

DEFINITION Returns the Cosine of the expression in radians.



$$\cos(A) = \frac{X}{H} \quad \text{or} \quad X = H \cdot \cos(A)$$

EXAMPLE Using COS in an expression will force ZBasic to calculate that expression in floating point. COS is a scientific function. You may configure BCD scientific accuracy separately for both Double and Single Precision immediately after loading ZBasic.

Integer Cosine may be accomplished with the predefined ZBasic USR function; USR9(angle in Brads). This returns the integer cosine of an angle in the range +-255 (corresponding to +-1). The angle must be in Brads. This example program will draw a sine wave using USR9:

```
MODE7 :CLS
FOR I=0 TO 255
  PLOT I<<2,-USR9(I)+384
NEXT I
```

For more information about scientific functions and derived math functions see the "Math" section of this manual. See CIRCLE for more about BRADS. Also see ATN, SIN,TAN,EXP,SQR.

CSRLIN function

FORMAT **CSRLIN**

DEFINITION Returns the line where the cursor is positioned.

EXAMPLE CLS
 PRINT
 PRINT
 PRINT CSRLIN

RUN

2

REMARK See POS to determine the horizontal cursor position.



Not supported with the Apple // or Z80 versions of ZBasic. For Apple // use PEEK(37) to get the current cursor line.

FORMAT CVB (string)

DEFINITION Returns the binary floating point value of the first n characters of the condensed number in *string* (depending on whether Single or Double Precision is used).

Double Precision Returns the digits of accuracy defined in configure for double precision. (default is 8 digits i.e. the first 8 string characters.)*

Single Precision Returns the digits of accuracy defined in configure for single precision. (default is 4 digits i.e. the first 4 string characters.)

This function is the compliment of MKB\$.

EXAMPLE

```
A#=12345.678: B!=12345.678
:
A$=MKB$(A#): B$=MKB$(B!)
PRINT LEN(A$), LEN(B$)
:
C#=CVB(A$): D!=CVB(B$)
PRINT C#, D!
```

RUN

```
8          8
12345.678  12345.7
```

REMARK This function is used with some versions of BASIC to save space on disk when storing large amounts of numeric data in strings with FIELD. ZBasic does this automatically but CVB is still useful for string packing, etc. Also see MKI\$,CVI,MKB\$, READ# AND WRITE#. This command is not compatible with CVS or CVD.

A few things to remember concerning CVB:

Null strings or 1 character strings return 0

Two character strings will return 2 digits of accuracy. Four character strings will return four digits. See "Floating Point Variables" for more information.



*See "Floating Point Variables" for detailed information on how extended double precision variables are stored and the added range of this precision for the Mac.

CVI function

FORMAT CVI (string)

DEFINITION Returns the binary integer value of the first 2 characters of string.

This function is the compliment of MKI\$.

EXAMPLE

```
A$=MKI$(30000)
PRINT LEN(A$)
:
Z%=CVI(A$)
PRINT Z%
END
```

RUN

```
2
30000
```

REMARK Also see MKI\$,CVB,MKB\$,READ# AND WRITE#.

A few things to remember concerning CVI:

Null string returns 0

One character strings will return the ASCII value.

Two character strings will return an integer value.

$ASC(\text{second character}) * 256 + ASC(\text{first character})$

This function was used with MBASIC to save space on disk when storing large amounts of numeric data. ZBasic does this automatically when using WRITE# and READ# but CVI is still useful for string packing, etc.



See DEFSTR LONG in the Mac appendix for using this function with LongIntegers. When LongIntegers are used the memory requirements are four bytes instead of two bytes. MSB and LSB are stored in reverse order for regular integers with this version.

FORMAT DATA *data item* [, *data item* [, ...]]

DEFINITION The DATA statement is used to hold information that may be read into variables using the READ statement. DATA items are a list of string or numeric constants separated by commas and may appear anywhere in a program.

No other statements may follow the DATA statement on the same line.

Items are read in the order they appear in a program. RESTORE will set the pointer back to the beginning of the first DATA statement. RESTORE n will set the pointer to the nth DATA item.

EXAMPLE

```
DATA Tom, Dick, Harry, 12.32, 233
READ A$, B$, C$, A#, B%
:
DEF TAB 6
PRINT "DATA items are: ";A$,B$,C$,A#,C%
```

RUN

```
DATA items are: Tom Dick Harry 12.32 233
```

```
DATA Tom, Dick, Harry, 12.32, 233
:
RESTORE 3
READ Name$
:
PRINT "Third DATA item is: ";Name$
```

RUN

```
Third DATA item is: Harry
```

REMARK Alphanumeric string information in a DATA statement need not be enclosed in quotes if the first character is not a number, math sign or decimal point.

Leading spaces will be ignored (unless in quotes). DATA statements can be included anywhere within a program and will be read in order.

Typical storage requirements for DATA items:

Number with zero value	2 bytes
Non-zero integer	3 bytes
Strings	Length of string + 2
Floating Point BCD	"See Floating Point Constants"
Floating Point Binary	"See Floating Point Constants"

See READ, PSTR\$ DIM and RESTORE for common statements used with DATA.

Note: See PSTR\$ for extremely efficient way of retrieving strings in DATA statements.



FORMAT

DEFINT *letter [- letter] [, letter [- letter],...]*
DEFSNG *letter [- letter] [, letter [- letter],...]*
DEFDBL *letter [- letter] [, letter [- letter],...]*
DEFSTR *letter [- letter] [, letter [- letter],...]*
***DEFDBL INT** *letter [- letter] [, letter [- letter],...]*

DEFINITION These statements define which variable type ZBasic will assume when encountering a variable name with letter as a first character and not followed by a type declaration symbol (% integer, ! single, # double, \$ string, & double integer).

DEFINT Integer
DEFSNG Single Precision
DEFDBL Double Precision
DEFSTR String
*DEFDBL INT LongInteger (Macintosh only)

ZBasic will assume that all variables are integers unless followed by a type declaration symbol or defined by a DEF type statement.

See "Configure" for another way of defining the default variable type.

letter Letter from A to Z. Case is not significant.
letter - letter Defines an inclusive range of letters.

EXAMPLE

DEFSNG A	<--- A and A! are the same variable (A\$ is still a string).
DEFDBL B	<--- B and B# are the same variable (B% is still an integer).
DEFINT F	<--- F and F% are the same variable (F! is still single prec.).
DEFSTR B-D, X, Y, Z	<--- B,C,D,X,Y and Z all strings
DEFDBL A, F-J, T	<--- A,F,G,H,I,J and T all Double precision
DEFSGL A, G, B-E	<--- A,G,B,C,D and E all Single Precision

REMARK Other versions of BASIC may assume all numeric variables are single precision unless otherwise defined. See the sections on "Floating Point Variables", "Math" and "Converting Old Programs" in the front of this manual for more information.



*Also see DEFSTR LONG in appendix for way of forcing HEX\$, OCT\$, UNS\$, CVI and MKI\$ to default to LongInteger instead of regular integer.

DEF FN statement

FORMAT **DEF FN** *name* [(*variable* [, *variable* [,...]])] = *expression*

DEFINITION This statement allows the user to define a function that can thereafter be called by FN name. This is a handy way of adding functions not provided in the language.

The expression may be a numeric or string expression and must match the type the FN name would assume if it was a variable name.

The name must adhere to variable name syntax.

The variable used in the definition of the function is a dummy variable. When using FN the dummy variables, other variables or expressions may be used to pass the values to the function. The variable should be of the right type used in the function.

EXAMPLE

```
DEF FN e# = EXP(1.)
DEF FN Pi# = ATN(1)<<2
DEF FN Sec#(x#) = 1.\COS(x#)
DEF FN ArcSin#(x#) = ATN (x# \ SQR( 1 - x# * x#))
:
PRINT FN Pi#
I#=4.2312
Planet# = FN ArcSin#(Sin(I#))* FN e# + FN Sec# (Ellipse#)
```

RUN

3.14159...

```
REM     A Handy rounding function
REM     Send the routine the number and places to round
:
DEF FN Round#(num#, places)=INT(num#*10^places+.5)/10^places
:
PRINT FN Round#(823192.12345675676,5)
X#=202031.12332
PRINT FN Round#(X#,2)
END
```

RUN

823192.12457
202031.12

REMARK One function may call another function as long as the function was defined first.

LONG FN is another form of DEF FN that allows multiple lines of code. It is very powerful for creating reusable subroutines.

See "Derived Math functions", "Functions and Subroutines", LONG FN, END FN and FN.

statement DEF LEN

FORMAT DEF LEN[=] number

DEFINITION The DEF LEN statement is used to reset the default length of string variables until the next DEF LEN statement is encountered. The number must be from 1 to 255.

If DEF LEN is not used string length default is 255 characters each. Each string will consume 256 bytes; 1 byte for length byte, the rest for characters.

Since strings will consume so much memory if their length is not defined; it is imperative that thought be given to string length, especially if memory is at a premium.

EXAMPLE

```
C$="Welcome"          <---Length of C$ defaults to 255 characters.
:
DEF LEN 20
DIM A$(10)           <---A$() allocated 20 characters per element.
Greeting$="Hello"    <---Greeting$ allocated 20 characters
:
DEF LEN 200
B$="Goodbye"         <---B$ allocated 200 characters
:
DIM 50 Z$            <---Z$ allocated 50 characters. See DIM
```

REMARK DEF LEN will allocate the specified amount of memory to every string that is defined after it (unless defined differently in DIM or another DEF LEN).

Strings that appear before the DEF LEN statement are not affected. For example, in the above program, C\$ is allocated the default length of 255 characters because it appeared BEFORE the DEF LEN statement.

DIM may also be used to set the length of string variables. See DIM.

Also see "String Variables" and "Converting Old Programs" in the front section for important information about strings and how they use memory.



Important Note: Always allocate one extra character for strings used with INPUT. Never use a one character string for INPUT. The extra character position is needed for the carriage return.

DEF MOUSE statement

FORMAT **DEF MOUSE** [=] *expression*

DEFINITION The DEF MOUSE statement is used to define the device to be used with the MOUSE functions and statements, or the type of mouse commands to use with the program.

DEF MOUSE=0 Regular ZBasic MOUSE commands for a mouse device. See MOUSE in this reference section.

MSDOS: Uses Microsoft™ compatible mouse devices. Be sure to "Configure" ZBasic for a mouse.

Apple //: Assumes a mouse is connected.

Macintosh: Standard MOUSE commands in this section of the reference manual. See DEF MOUSE=1 to do MSBASIC type mouse commands.

Z80: NOT SUPPORTED.

DEF MOUSE=n Tells ZBasic that other devices are to be used instead of a MOUSE (in the case of the Macintosh it tells ZBasic to use MSBASIC mouse syntax).

MSDOS: n=1 defines joystick/paddle A*
 n=2 defines joystick/paddle B*
 n=3 defines a lightpen device

Apple //: n=1 defines a joystick/paddle device*

 *Mouse(3) function returns button status:
 0=No button pressed
 1=Button zero pressed
 2=Button one pressed
 3=Both buttons pressed

Macintosh: n= non-zero sets commands to MSBASIC mouse commands. See Macintosh appendix for specifics.

Z80: NOT SUPPORTED.

EXAMPLE See the appendix for your computer for specifics.

REMARK See MOUSE in this reference section and in your appendix for specifics.



MOUSE or DEF MOUSE is not supported with any Z80 versions of ZBasic. This is due to the fact that most Z80 computers do not offer this hardware device.

statement DEF TAB

FORMAT DEF TAB [=] *expression*

DEFINITION The DEF TAB statement is used to define the number of characters between tab stops for use in PRINT,PRINT# or LPRINT statements

Tab stops are the number of spaces to move over when the comma is encountered in a PRINT statement.

The expression must be a number from 1 to 255. TAB default is 16.

EXAMPLE

```
PRINT 1,2,3          <---Tab stop default is 16, 32, 48...
DEF TAB = 8          <---Tab stops now set to 8, 16, 24...
PRINT 1,2,3: PRINT
:
FOR X=1 TO 5
  DEF TAB=X
  PRINT 1,2,3
NEXT X
```

RUN

```
1           2           3
1      2      3
1 2 3
1  2  3
1   2   3
1    2    3
1     2     3
```

REMARK Also see TAB,WIDTH,WIDTH LPRINT and PAGE.

DEF USR statement

FORMAT **DEF USR** *digit = expression*

DEFINITION The DEF USR statement is used to define the addresses of up to 10 machine language user subroutines; USR0 to USR9.

EXAMPLE *Examples only. Do Not Use!*

```
REM Calls graphic routine at memory address 5000
DEFUSR1=5000
X=USR0(45)
:
DEFUSR2=23445
PRINT USR2(x)
```

REMARK A machine language return is needed at the end of the routine to return program control to ZBasic.

See USR, MACHLG, CALL, LINE, VARPTR, BIN\$, HEX\$, OCT\$, UNS\$, PEEK, PEEKWORD, POKE, POKEWORD and the chapter "Machine Language".

Some other default USR functions are included in the appendix for your computer.



Warning: Use of this command requires a knowledge of machine language and a computer's hardware. Porting of programs with this statement may not be possible without re-writing the routines.

statement DELAY

FORMAT DELAY *expression*

DEFINITION The DELAY statement will cause a program to pause a specified amount of time.

The expression sets the delay in milliseconds; thousandths of a second.

EXAMPLE

```
CLS
FOR I = 1 TO 5
  PRINT "DELAYING ";I;"SECONDS"
  DELAY I * 1000
NEXT I
END
```

RUN

```
DELAYING 1 SECONDS
DELAYING 2 SECONDS (after 1 second)
DELAYING 3 SECONDS (after 2 second)
DELAYING 4 SECONDS (after 3 second)
DELAYING 5 SECONDS (after 4 second)
```

```
FOR X=1000 TO 0 STEP -50
  PRINT X
  DELAY X
NEXT
```

(try it)

REMARK

The <BREAK> key is not scanned during DELAY. Any negative expression will cause delays in excess of 32 seconds (the unsigned value). Note that DELAY -1 will delay over 65 seconds (unsigned -1 = 65,535).

There may be a slight time variation from machine to machine due to processor speed, interrupts, hardware differences, etc.

Also see DATE\$ and TIME\$.



Also see TIMER.

DELETE command

FORMAT **DEL [ETE] *line***
 DEL [ETE] *-line*
 DEL [ETE] *line - line*
 DEL [ETE] *line-*

DEFINITION This command will remove a line or range of lines from a program in memory.

DELETE is used from the Standard Line Editor.

EXAMPLE

```
10 CLS
20 FOR I = 1 TO 10
30 PRINT "NUMBER "; I
40 NEXT I
50 END

DEL 10-20

LIST

30 PRINT "NUMBER "; I
40 NEXT I
50 END

-----

10 "FRED" PRINT "NUMBER ";I
20 PRINT "Fred was here"
30 END

DELETE "FRED"

LIST

20 PRINT "Fred was here"
30 END
```

REMARK Use this command with care as recovery of deleted lines is not possible.

FORMAT **DIM** [*len*] *var* [*type*] [(*number* [, *number* ..])][, ...]

DEFINITION The DIM statement is used to allocate memory for variables and array variables and to define common variables for chained programs.

len Defines the length a of a string (how many characters it may hold). This is optional and defines the length of all the following string variables in that DIM statement or until a new length is encountered in that statement. The default is 255 characters unless changed by a previous DEFLEN.

var The name of a variable (any variable type).

type Forces the *variable* to be of that *type*.

- %=Integer
- &=LongInteger (Macintosh only)
- !=Single Precision
- #=Double Precision
- \$=String

number Also see "Variables" in the front section of this manual.
The maximum number of elements that a dimension may contain from 1 to 32,767 elements (add one if array BASE option is set to zero. default=0). Only numbers may be used, not variables.

EXAMPLE See the following page for more information and examples.

REMARK Use care when allocating memory with the DIM statement.

See BASE OPTION,DEFLEN,"Array Variables", "String Variables", INDEX\$ and RUN+ for more important information about using DIM.



Macintosh: This version is limited to 2,147,483,648 elements in an array.
MSDOS: In order to optimize performance; integer variables and integer array variables are limited to one 64k segment. String and BCD arrays may cross segment borders to use up to available memory.

continued next page...

DIM statement

DIM continued

DETERMINING THE MEMORY NEEDS OF DIMMED ARRAYS

```
DIM A%(10,10,10), A#(5), A!(9,7), B$(10), 5Cool$(20
DIM Long&(10): REM Macintosh Only
```

The following chart shows how to calculate the memory requirements of the arrays dimensioned above with a BASE OPTION of zero.

<u>ARRAY</u>	<u>TYPE</u>	<u>Bytes per Element</u>	<u>How to Calculate**</u>	<u>Memory Required</u>
A%(10,10,10)	Integer	2	11*11*11*2	2662
A#(5)	Double Precision	8	6*8	48
A!(9,7)	Single Precision	4	10*8*4 320	
B\$(10)	String	256	11*256 2816	
Cool\$(20)	String	6	21*6	126
Long&(10)	LongInteger	4	11*4	44

DEFINING STRING LENGTHS WITH DIM

```
DIM X$(10), 20A$, Z$(5), 45TEST$, 10MD$(20,20)
```

In the example above the maximum character capacities are:

X\$ 255 (default is 255)
A\$ 20
Z\$(5) each element if Z\$ as 20* (21*5=105 total bytes)
TEST\$ 45
MD\$(20,20) each element of MD\$(20,20) as 10.
 (20*20*11=4400 total bytes of memory used)

* If no length is defined, the last given length in that DIM statement is used. In the example each element of Z\$(n) gets a length of twenty. If no length is defined in that DIM statement then 255 characters is the default (or the last length used in DEF LEN).

**If you configure BASE OPTION 1 you will not need to add one to the dimension. To calculate the memory required for A%(10,10,10): 10*10*10*2. See "Configure".

Note: Add one to the defined length of each string for the length byte to determine the actual memory requirement of the string. This extra byte is the "Length byte" and it is the first byte in the string. It is what is pointed at by VARPTR(var\$).



Important Note: Unpredictable system errors may result if an attempt is made to assign a string variable a string longer than its allocated length. It is also important to define the length of a string at least one greater than the maximum number of characters received in an INPUT or LINEINPUT statement.

FORMAT DIR[*drivespec*]

DEFINITION DIR will display the directory of the disk drive specified by *drivespec*.

The *drivespec* will vary from one computer to the next. See your Computer's Disk Operating System reference manual for syntax.

EXAMPLE DIR <ENTER>

```
LEDGER.COM      MAY.LEDJUN.LED
JUL.LEDAUG.LED
```

ZBasic Ready

REMARK The appearance of the directory layout will vary by computer. See appendix for further information. This is a command so it does not operate during runtime.

See below, or your appendix, for possible ways of getting directories at runtime.



Macintosh: Syntax is DIR "rootname or foldername". To get a directory during runtime see FILE\$ in the appendix. LDIR will output the directory to a printer.

MSDOS: Use DIR *.BAS to see all the .BAS files or DIR Z*.* to see all the files starting with Z. To get a directory during runtime see FILES.

Apple ProDOS: To get a directory during runtime; OPEN"|" the directory pathname. Example: OPEN"|"1,"ZBASIC". See directory layout in ProDOS reference manual for more information about directory file layout. This version also supports LDIR to list the directory to the printer. CAT may be used as well as DIR.

Apple DOS 3.3: To get a directory during runtime:

```
LONG FN DIR (slot,drive)
  POKE &AA6A,slot
  POKE &AA68,drive
  CALL &A56E
END FN
```

Z-80: See appropriate section in appendix for your computer and DOS. Some Z80 versions do not allow getting a directory at runtime.

DO statement

FORMAT DO

```
.  
. UNTIL expression
```

DEFINITION The DO statement is used to define the beginning of a loop with the UNTIL statement defining the end.

Program functions and statements appearing between the DO and UNTIL will be executed over and over again until the expression defined at the UNTIL statement is TRUE.

EXAMPLE

```
DO  
  PRINT"Hi!"  
UNTIL LEN(INKEY$)  
END
```

RUN

```
Hi!  
Hi!  
Hi!  
Hi!          <-----You press a key and it stops
```

```
DO  
  X=X+1  
UNTIL X=2492  
PRINTX  
END
```

RUN

2492

REMARK The statements in a DO loop will be executed at least once. See WHILE-WEND for a loop type that ends immediately if the condition is false.

ZBasic automatically indents text appearing between a DO and UNTIL two spaces. This is helpful in debugging and documenting programs.

See the "Structure" and "Loops" sections of this manual for more information.

Also see FOR-NEXT-STEP and WHILE-WEND.

FORMAT **E** *line*
EDIT *line*

DEFINITION EDIT is used from the Standard Line Editor to specify the line you wish to edit.

EDIT may be abbreviated to E. A comma will start editing at the line currently selected by ZBasic's line pointer. List of the EDIT sub-commands:

<u>SUB-COMMAND</u>	<u>DEFINITION</u>
[n]<SPACE>	- MOVE CURSOR RIGHT (n characters)
[n]<BACKSPACE>	- MOVE CURSOR LEFT (n characters)
I	- Begin INSERT mode at cursor position
X	- Goto the end of the line and EXTEND it
<ESC>	- Exit INSERT mode (you will still be in line edit mode)
[n]D	- DELETE characters (if n is used deletes n characters)
[n]C key	- CHANGE character to <key> [n] times
H	- HACK to end of line and enter INSERT
[n]S key	- SEARCH for [n]the occurrence of <key>
L	- LIST line being edited, home cursor
A	- ABORT changes, restore original line
[n]K key	- KILL text to [n]the occurrence of <key>
<ENTER>	- EXIT editing with changes intact
<BREAK>	- ABORT EDIT SESSION (no changes made)

Note: n is a number from 1 to 255. If n is not used, one is assumed.

EXAMPLE 10 FOR I = 1 TO 20
 20 PRINT I
 30 NEXT I

EDIT 20 <---- or E20 (comma if 20 was the last line used.)

20 _ <---- Press spacebar or backspace to move cursor.
 Use keys above to edit this line.

REMARK If you want to edit the current line, press the comma key <,> in command mode. It will do the same as E <ENTER>.

Line numbers may be edited in ZBasic. The line being edited will remain unchanged, the edited line with the new line number will be created.

See the "Standard Line Editor" section in the beginning of this manual.

Also see FIND,DELETE,AUTO and LIST.



These versions offer full screen editors as well as the Standard Line Editor. See "Full Screen Editor" in the appropriate appendix for details.

ELSE statement

FORMAT IF-THEN-ELSE *line or label*
IF-THEN-ELSE *statement(s)*

DEFINITION ELSE is used with an IF statement to route control on a false condition.

ELSE may refer to a linenummer or label or it may be followed by one or more statements that will be executed if the condition in the IF statement is FALSE.

EXAMPLE X=99
IF X = 100 THEN STOP ELSE PRINT X
END

RUN

99

```
IF X=100 THEN STOP ELSE "End"  
END  
:  
PRINT"Stopped here."  
END
```

RUN

Stopped here.

REMARK All statements on a line following an ELSE are conditional on that ELSE.

See "Structure",IF-THEN,LONG IF,XELSE and ENDIF.



Also see SELECT CASE.

FORMAT **END**

DEFINITION END is used to stop the execution of a program.

END will return control to the Standard Line Editor if program was executed using RUN, or to the operating system if the program was compiled using RUN* or RUN+.

EXAMPLE PRINT "HELLO"
 END
 PRINT "THERE"

RUN

HELLO

REMARK END will close all open files.

Also see STOP and TRONB.



See SHUTDOWN.

END FN statement

FORMAT LONG FN

```
.  
. .  
END FN [= expression]
```

DEFINITION Marks the end of a LONG FN statement.

The optional expression **MUST** be numeric for numeric functions (#,%,&,!) and **MUST** be a string (\$) for string functions.

EXAMPLE

```
REM Removes spaces from the end of a string  
LONG FN RemoveSpace$(x$)  
  WHILE ASC(RIGHT$(x$,1)=32  
    x$= LEFT$(x$, LEN(x$)-1)  
  WEND  
END FN= x$  
Name$="ANDY"  
PRINT "Before:";Name$;"*"  
PRINT "After:"; FN RemoveSpace$(Name$);"*"
```

RUN

```
ANDY      *      ANDY*
```

```
REM Example of a simple Matrix Multiplication  
DIM A%(1000)  
:  
LONG FN MatrixMult%(number%, last%)  
  FOR temp%= 0 TO last%  
    A%(temp%)=A%(temp%)*number%  
  NEXT  
END FN  
:  
A%(0)=1: A%(1)=2:A%(2)=3  
FN MatrixMult%(10,3)  
PRINT A%(0), A%(1), A%(2)
```

RUN

```
10          20          30
```

REMARK

If an END FN is omitted in a LONG FN construct, a structure error will occur. You must exit a function from an END FN otherwise problems will occur internally.

Also see "Functions and subroutines", "Structure", LONG FN, FN statement, FN function and DEF FN.



Important Note: Loops like FOR-NEXT, DO-UNTIL or WHILE-WEND must be entirely contained within a LONG FN-END FN. Do not exit a function except at the END-FN.

FORMAT LONG IF *expression*
.
[XELSE]
.
END IF

DEFINITION This is an end marker for the LONG IF statement.

Program execution will continue normally at the END IF after completion of a LONG IF or XELSE.

EXAMPLE

```
Love$="Forever"  
LONG IF Love$="Forever"  
    PRINT "How Romantic!"  
XELSE  
    PRINT "How heartbreaking!"  
END IF  
END
```

RUN

How Romantic!

REMARK If an END IF is omitted in a LONG IF construct, a structure error will occur.

See "Structure",LONG IF,IF-THEN,ELSE and XELSE.



Also see SELECT CASE.

END SELECT statement

FORMAT SELECT [CASE] [expression]
 CASE [IS] relational condition1[,relational condition][,...]
 statement(s)
 CASE [IS] condition[,condition][,...]
 statement(s)
 CASE [IS] boolean expression
 statement(s)
 CASE ELSE
 statement [:statement:...]]
END SELECT

DEFINITION END SELECT is the end marker for the SELECT/CASE structure.

When SEIECT/CASE is encountered, the program checks the value of the controlling expression or variable, finds the CASE that compares true and executes the statements directly following the CASE statement. After these statements are performed, the program continues at the line after the END SELECT statement:

EXAMPLE A=100
 SELECT A
 CASE >100
 PRINT "A>100"
 CASE 100
 PRINT "A=100"
 CASE ELSE
 PRINT"None of the above"
 END SELECT
 PRINT "Program continues..."
 END

RUN

A=100
Program continues...

REMARK Also see SELECT and CASE.



SELECT CASE is not supported with the Z80 versions. See IF and LONG IF for accomplishing the same thing.



SELECT CASE is not supported with this version. See IF and LONG IF for accomplishing the same thing.

FORMAT EOF (*filenumber*)

DEFINITION Returns true if end-of-file condition exists for *filenumber*, returns zero if the end-of-file has not yet been reached. This function is only available on the Macintosh and MSDOS versions of ZBasic.

EXAMPLE

```
OPEN"1",1,"FILE.TXT"
DO
  LINEINPUT#1, A$
  PRINT A$
UNTIL EOF(1)
CLOSE#1
END
```

What to do if you don't have EOF on your computer.

```
ON ERROR GOSUB 65535 <--- Enable disk error trapping
OPEN"1",1,"FILE.TXT"
IF ERROR GOSUB"Error message"
DO
  LINEINPUT#1, A$
  PRINT A$
UNTIL ERROR <>0
IF ERROR <> 257 THEN GOSUB "Error message"
ERROR=0 <---Error 257 is an end-of-file error. Reset Error here then continue.
CLOSE#1
END
:
"Error message"
PRINT "A disk error occurred: "; ERRMSG$(ERROR)
INPUT"<C>ontinue or <S>top? ";temp$
If temp$="C" THEN ERROR=0:RETURN
STOP
```

REMARK Some versions of ZBasic do not support EOF because of system reasons. Also see ERROR function and statement, ON ERROR and ERRMSG\$



EOF is not supported on Z80 versions of ZBasic. Use the second example above to accomplish the same thing.



EOF is not supported on the Apple // ProDOS or DOS 3.3 versions of ZBasic. Use the second example above to accomplish the same thing.

ERRMSG\$ function

FORMAT ERRMSG\$ (*expression*)

DEFINITION Returns the error message string for the error number specified by expression. In most cases you will use the number returned by the ERROR function when a disk error has occurred.

EXAMPLE

```
OPEN "I",1, "OLDFILE"
ON ERROR GOSUB "Error message"
.
.
.
"Error message"
PRINT "A disk error has occurred!!"
PRINT "The error was: ";ERRMSG$(ERROR)
ERROR=0:REM ALWAYS SET ERROR TO ZERO AFTER ERROR OCCURS!
RETURN
```

RUN

```
A disk error has occurred!!
The error was: File Not Found Error in File #1
```

```
FOR X=0 TO 255
  PRINT ERRMSG$(X)
NEXT X
```

RUN

PRINTS ALL THE ERROR MESSAGES FOR THAT COMPUTER.

REMARK ZBasic will display disk errors for you unless you use the ON ERROR disk trapping options.

The ERROR function is commonly used for error trapping and display purposes. The expression is stored as follows:

The low byte is used for the ERROR number (ERROR AND 255)
The high byte is used for the file number (ERROR>>8) or (ERROR/256)

See "Disk Errors", ON ERROR GOSUB and ERROR functions and statements.

function ERROR

FORMAT ERROR

DEFINITION Returns the number of an ERROR condition, if any.

Zero (0) is returned if no error has occurred.

This function is available to programmers who wish to trap disk errors using the ON ERROR statement.

EXAMPLE

```
ON ERROR GOSUB 65535:REM User disk trapping enabled
OPEN "I",1,"OLDFILE"
IF ERROR=259 GOSUB"NOT FOUND" GOTO 20
ON ERROR RETURN: REM Let ZBasic do the error checking now!
.
.
.
"NOT FOUND"
REM ERROR 259 is: File Not Found error in Filenumber 1
PRINT" The file is not on that disk!"
PRINT" Please insert the correct disk"
PRINT" and press <ENTER>"
INPUT A$:ERROR=0:RETURN
```

REMARK ERROR may also be used as a statement. See ERROR statement, ERRMSG\$ and ON ERROR GOSUB.



Important Note: If you do the disk error trapping, ERROR must be reset to zero after a disk error occurs or ERROR function will continue to return an error value.



Macintosh: Also see SYSERROR in appendix.

MSDOS: See appendix for ways of doing critical error handling.

Apple ProDOS: See appendix for additional ways of trapping ProDOS errors.

ERROR statement

FORMAT **ERROR** [=] *expression*

DEFINITION Allows the programmer to set or reset ERROR conditions for the purpose of disk error trapping.



Important Note: If you do the disk error trapping, ERROR must be reset to zero after a disk error occurs or ERROR function will continue to return an error value.

EXAMPLE

```
REM This routine checks to see if a file exists. If it
REM does exist it is opened as random, if it doesn't
REM exist an error message is returned.
:
LONG FN Openfile%(files$, filenum%, reclen%)
ON ERROR GOSUB 65535: REM Disk error trapping on
"Open file"
OPEN"I",filenum%,file$
LONG IF ERROR
LONG IF (ERROR AND 255) <>3
PRINT@(0,0);"Could not find: ";file$;" Check drive"
INPUT"and press <ENTER> when ready";temp%
ERROR=0: GOTO "Open file"
END IF
XELSE
CLOSE# filenum%
END IF
ON ERROR RETURN: REM Give error checking back to ZBasic
OPEN"R",filenum%, file$, reclen%
END FN
```

REMARK ERROR may also be used as a function. See "Disk Error Trapping", ERROR function, ERRMSG\$ and ON ERROR.



Macintosh: Also see SYSERROR in appendix.

MSDOS: See appendix for ways of doing critical error handling.

Apple ProDOS: See appendix for additional ways of trapping ProDOS errors.

FORMAT EXP (*expression*)

DEFINITION Returns e raised to the power of expression. This function is the compliment of LOG. The BCD internal constant of the value of e is:

2.71828182845904523536028747135266249775724709369995957

The result will be rounded to the digits of precision configured for Double Precision accuracy.

EXAMPLE

```
DEFDBL A-Z
DO
  INPUT "ENTER A NUMBER ";X
  PRINT "e RAISED TO X =" ; EXP(X)
UNTIL X=0
END
```

RUN

```
ENTER A NUMBER _ 1
e RAISED TO X = 20718281828459 <--- 14 digit accuracy
```

REMARK This is a scientific function. See "Configure" for information about configuring scientific accuracy.

For more information about scientific functions see "Math", "Math expressions", "Floating Point Variables", COS,SIN,ATN,TAN,SQR and raise to the power"A".

FILL statement

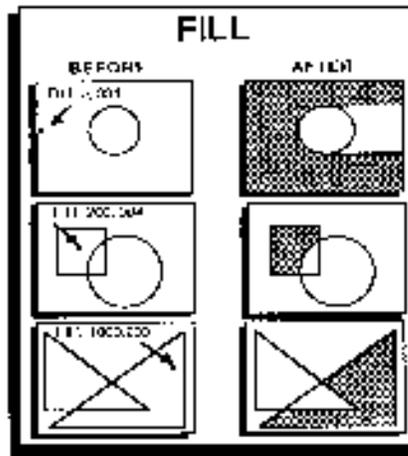
FORMAT FILL *expression_x*, *expression_y*

DEFINITION The purpose of FILL is to paint an area of the screen in the current COLOR. The point defined by the two expressions are:

expression_x (horizontal position) and *expression_y* (vertical position).

Fill will search for the uppermost point in the contained area that has the background color, then start filling from left to right and down. For this reason irregular shapes may not fill completely with one fill command. It may be necessary to use a fill statement for each appendage.

EXAMPLE



```
COLOR=1  
FILL 0,284
```

RUN

See chart.

REMARK FILL may not be available on machines without the capability of seeing pixels on the screen. See computer appendix. Also see CIRCLE FILL, BOX FILL, POINT and PLOT.



BOX FILL, CIRCLE FILL and the QuickDraw routines like FILLPOLY, FILLRGN, FILLRECT etc. are much faster ways of filling areas.

command FIND

FORMAT	FIND	<i>commands or keywords</i>
	FIND #	<i>line</i>
	FIND "	<i>quoted string text or labels</i>
	FIND REM	<i>items in REM statements</i>
	FIND DATA	<i>items in DATA statements</i>

DEFINITION FIND is used in the Standard Line Editor to locate text in a program.

To FIND additional occurrences, press semi-colon (;) or FIND <ENTER>.

EXAMPLE	YOU TYPE	ZBASIC FINDS
	FIND "HELLO	01010 A=20:PRINT"HELLO THERE"
	FIND A\$	01022 Z=1:A\$=B\$:PRINTA\$+B\$
	or...	01222 BA\$="hello"
	or...	01333 ABA\$="goodbye"
	FIND 99	05122 F=2:X=X+2+F/999
	FIND #12345 (line#)	08000 GOTO 12345
	FIND X(C)	03050 A=1:T=ABS(X(C)/9-293+F)
	or...	03044 ZX(C)=4
	FIND PRINT	00230 A=92:PRINTA
	FIND "SUB5	00345 "SUB500": CLS
	or...	03744 GOSUB "SUB500"
	FIND OPEN	03400 OPEN"R",1,"FILE54",23
	FIND CLOSE	09900 CLOSE#2
	FIND REM This	02981 REM This is a remark
	FIND DATA 123, 232	09111 DATA 123, 232
	FIND DATA "Fred"	10233 DATA "Tom", "Dick", "Fred"

REMARK When finding a string inside quotes, you must supply all of the characters up to the point that will insure the uniqueness of the string.

See "Standard Line Editor" in the beginning of this manual.



See "Full Screen Editor" in the appropriate appendix for other FIND commands.

FIX function

FORMAT **FIX** (*expression*)

DEFINITION Truncates the digits on the right side of the decimal point.

EXAMPLE `PRINT FIX (123.456),`
`A#=1293.21`
`PRINT FIX(A#),`
`PRINT FIX (.12340),`
`PRINT FIX (999999.455) + 0.`

RUN

123 1293 0 999999

REMARK FIX works the same as INT in ZBasic. They are both included to maintain compatibility with other forms of BASIC. FIX will consider an expression floating point.

FRAC is the opposite of FIX. It returns the fraction part of the number.

See FRAC and INT.

FORMAT FN *name* [(*expression*₁ [, *expression*₂ [, ...]])]

DEFINITION FN calls a function by name which was previously defined by DEF FN or LONG FN. The name of the function must follow the syntax of variable names, that is, a string FN must have a name with a \$, and integer FN must have a name with a %, etc.

The expressions must match the variable types as defined by the DEF FN or LONG FN. Numeric expressions are not a problem, string expressions allow only simple strings.

FN may not be used before it is defined with DEF FN or LONG FN.

EXAMPLE

```
DEF FN e# = EXP(1.)
DEF FN Pi# = ATN(1) << 2
DEF FN Sec#(x#) = 1. \ COS(x#)
DEF FN ArcSin#(x#) = ATN (x# \ SQR(1-x# * x#))
:
PRINT FN Pi#
```

RUN

3.14159... <---Returned in the current digits of accuracy

```
REM Round number to the number of places indicated.
LONG FN ROUND#(number#, places)
    number# = INT(number# * 10^places + .5) / 10^places
END FN = number#
:
PRINT FN ROUND#(43343.327, 2)
```

RUN

43343.33

REMARK This function is useful for saving program space and for making a program easier to read.

Also see "Functions and Subroutines", "Structure", LONG FN, END FN, DEF FN, APPEND and FN statement.

FN statement

FORMAT FN *name* [(*expression*₁ [, *expression*₂ [, ...]])]

DEFINITION FN calls a function by name which has previously been defined by a DEF FN or a LONG FN.

The expressions must match the variable types as defined by DEF FN or LONG FN.

EXAMPLE

```
DEF FN LastChr%(x) = PEEK( x + PEEK(x))
LONG FN RemoveSpace$(x$)
  WHILE FN LastChr$(VARPTR(x$)) = ASC(" ")
    x$= LEFT$(x$, LEN(x$)-1)
  WEND
END FN= x$
Name$="ANDY      "
PRINT Name$; "*" , FN RemoveSpace$(Name$); "*"
```

RUN

```
ANDY      *          ANDY*
```

REMARK Also see "Functions and Subroutines", "Structure", LONG FN, END FN, DEF FN, APPEND and FN function

FORMAT **FOR** *variable* = *expression*₁ **TO** *expression*₂ [**STEP** *expression*₃]
 .
 .
 .
NEXT[*variable*][, *variable* ...]

DEFINITION Permits the repeated execution of commands within the loop.

A FOR/NEXT loop will automatically increment variable by the amount set by STEP and compare this to the end value, expression₂, exiting the loop when var exceeds this value after adding STEP. Default STEP = 1.

Note the loop will be executed at least once with the value of expression₁.

EXAMPLE **FOR** Counter = 0 **TO** 100 **STEP** 2
 PRINT Counter;
 NEXT

RUN

0 2 4 6 8 10 12 ... 100

FOR Counter = 100 **TO** 0 **STEP** -2
 PRINT Counter;
 NEXT Counter

RUN

100 98 96 94 92 90 88 ... 0

FOR Counter# = 0.0 **TO** 1.0 **STEP** .01
 PRINT Counter#;
 NEXT Counter#

RUN

0 .01 .02 .03 .04 ... 1

REMARK ZBasic will automatically indent all of its loop structures in listings. This is helpful in debugging and documenting programs.

See chapter called "Loops" and WHILE-WEND and DO-UNTIL.

Note: If STEP is set to zero, the program will enter an endless loop. If the variable is an integer, do not allow the loop to exceed 32,767 or you will enter an endless loop (unsigned integer).

FRAC function

FORMAT **FRAC** (*expression*)

DEFINITION FRAC returns the fractional part of expression. The digits to the left of the decimal point will be truncated.

This function is the complement of INT and FIX.

EXAMPLE A#=123.456
 B#=99343.999
 C#=3.5
 :
 PRINT A#, FRAC(A#)
 PRINT B#, FRAC(B#)
 PRINT C#, FRAC(C#)
 PRINT 2.321, FRAC(2.321)

RUN

```
123.456.456
99343.999   .999
3.5         .5
2.321      .321
```

REMARK This function will automatically set floating point calculation.

FIX and INT are the opposite. They return the whole part of the number.

See FIX and INT.

FORMAT GET (x1,y1)-(x2,y2), variable[array(index[, index...,])]

DEFINITION Stores a graphic image from the screen into a variable or variable array so that it may be retrieved later and put to the screen with PUT.

GET and PUT are extremely fast and useful for sophisticated graphic animation.

x1,y1 Coordinates of the upper-left-corner of the graphic image on the screen.
x2,y2 Coordinates of the lower-right-corner of the image.

Coordinates are pixel coordinates; use with COORDINATE WINDOW.

The image is normally stored in memory specified by an integer array since it is easier to calculate how much memory is required this way (although other variables may also be used as long as the memory set aside is correct).

To calculate the amount of bytes to DIM for a graphic image, use this equation. Bits-per-pixel (bpp) has to do with colors or grey levels available. See next page for specifics:

$$6 + ((y2 - y1) + 1) * ((x2 - x1) + 1) * \text{bpp} + 7 / 8$$

Failure to DIM enough memory for an image will cause unpredictable system errors so be sure to carefully calculate the memory needed.

EXAMPLE

```

DIM A(750)                <---Bytes above divided by two for integer array
MODE 7                   <---Not needed on the Macintosh version
COORDINATE WINDOW       <---Pixel coordinates
:
CIRCLE 100,100,80
GET (0,0)-(100,100), A1
:
FOR x= 1 TO 200 STEP 3
  PUT (x, 90), A(1)      <---Does twice to move the image across
  PUT (x, 90), A(1)      the screen without disturbing the background
NEXT x
:
END

```

This routine moves a section of a circle across the screen. It is PUT to the screen twice so the item doesn't repeat and it will appear to move across the screen without disturbing the background (default PUT mode is XOR).

continued...

GET statement

REMARK

Important Note: Failure to DIM enough memory for the variables storing the graphic images may result in unpredictable system problems.

Also see DIM and PUT.



Macintosh: With this version of ZBasic, PUT has another, optional, parameter: PUT (x1,y1) [-(x2,y2)], var. The second parameter allows you to scale the image, making it either larger or smaller by giving the rectangle size in which it is to appear. The x2,y2 parameter is the lower-right corner of the image.

Bits-per-pixel (bpp) will vary by the type of Macintosh you have. The standard black and white Macintoshes have one bit per pixel.

The Macintosh II may have up to 32 bits-per-pixel. Sixteen colors is 4 bpp, 256 colors is 8 bpp. Check addendum or "Inside Macintosh Volume V (Color Quickdraw)" for the specifics of your color board.



MSDOS: Bits per pixel (bpp) will vary by the graphics adaptor board being used:

<u>TYPE</u>	<u>MODE(s)</u>	<u>COLORS</u>	<u>BITS PER PIXEL (bpp)</u>
CGA	5	4	2
CGA	7	2	1
EGA	16-19	3-16	2 (64K or less on EGA card)
EGA	16-19	16	4 (More than 64K on card)
HERCULES	20	1	1



Z80: GET and PUT are not supported with these versions of ZBasic.



Apple // ProDOS and DOS 3.3: GET and PUT are not supported with these versions. See DRAW example on ProDOS disk and the BLOAD and BSAVE functions for possible alternatives.

FORMAT **GOSUB** *line or label*

DEFINITION GOSUB will call that part of a program starting with line or label and return to the next statement following the GOSUB when RETURN is encountered.

EXAMPLE

```

10 GOSUB 40: PRINT "All Done!"
20 END
30 :
40 PRINT"Hello"
50 RETURN
    
```

RUN

```

HELLO
All Done!
    
```

```

GOSUB "Hello Routine"
PRINT "All Done!"
END
:
"Hello Routine"
PRINT "Hello"
RETURN
    
```

RUN

```

HELLO
All Done!
    
```

REMARK On multiple statement lines, a RETURN will return control to the next statement on the line following the originating GOSUB.

To avoid errors, be certain there is a line with the number or label that you GOSUB. All subroutines must be terminated with a RETURN statement.

Note: If ZBasic encounters a RETURN without a matching GOSUB, it will return to the operating system or the editor. ZBasic does not check for stack overflow which may cause errors if subroutines do not end with a RETURN.

See RETURN LINE,GOTO, ON GOTO and ON GOSUB.



See SEGMENT RETURN in appendix.

GOTO statement

FORMAT **GOTO** *line or label*

DEFINITION **GOTO** will transfer control to a line or label in a program.

Note that excessive use of this statement is considered inappropriate for structured code because in complex programs it becomes extremely hard to read.

In most programming situations GOSUB, DO-UNTIL, WHILE-WEND, FOR-NEXT or other programming structures are much easier to follow.

EXAMPLE 10 X=X+1
 PRINT X,
 20 IF X<5 THEN GOTO 10

RUN

1 2 3 4

```
"Loop"  
X=X+1  
PRINT X,  
IF X<5 THEN GOTO "Loop"
```

RUN

1 2 3 4

REMARK A line error will occur during compile if the destination line or label cannot be found.

See "Structure", GOSUB,ON GOTO,ON GOSUB,LONG FN,FN statement,WHILE, DO,FOR,LONG IF.

FORMAT **HELP** [*number*]

DEFINITION HELP without a number prints the HELP menu to the screen. This menu will give you corresponding numbers to the help topics available. This command is used from the Standard Line Editor.

Type HELP and a number to get answers to a specific topic.

Press the SPACE BAR to continue when you see "MORE".

EXAMPLE HELP

A menu for your version of ZBasic will be printed to the screen. To get help for an item in the menu, type HELP and the number corresponding to that item.

REMARK HELP will return control to the Standard Line Editor upon completion of the listing.

If the help file has been deleted from the disk a File Not Found Error will occur. Check your computer appendix for the filename of the HELP file.



The HELP window is brought up when you type this command or select "About ZBasic" under the  menu. The command does not work exactly as above. Just double click the appropriate item with the mouse.

HEX\$ function

FORMAT **HEX\$(expression)**

DEFINITION The HEX\$ function converts a numeric expression to a four character HEXadecimal string (BASE 16). The following program will convert a Decimal number to HEX or HEX to Decimal. Some sample HEX numbers:

<u>Decimal</u>	<u>Hexadecimal</u>
0-9	0-9
10	A
11	B
12	C
13	D
14	E
15	F

EXAMPLE DO
 INPUT"Decimal number to convert: ";Decimal%
 PRINT "Decimal";Decimal%;"= HEX ";HEX\$(Decimal%)
 PRINT
 :
 INPUT"HEX number to convert: ";Hx\$
 Hx\$="&H"+Hx\$
 PRINT"Decimal value of ";Hx%;"="VAL(Hx\$)
 PRINT"The unsigned Decimal value of "Hx%;"=" UNS\$(VAL(Hx\$))
 UNTIL (Decimal% =0) OR (LEN(Hx\$)=2)

RUN

```
Decimal number to convert: 255
Decimal 255= HEX FF

HEX number to convert: F9CD
Decimal value of F9CD = -1587
The unsigned Decimal value of F9CD = 63949
```

REMARK Floating point numbers will be truncated to integers.

See "Numeric Conversions", VAL, OCT\$, BIN\$ and UNS\$.



See DEFSTR LONG in the appendix for doing LongInteger conversions in Hex, Octal, CVI and MKI\$. In this case HEX\$ would return an eight character string.

FORMAT **IF** *expression* **THEN** *line [or label]* [**ELSE** *line [or label]*]
IF *expression* **THEN** *statement [:statement: ...]* [**ELSE** *statement [:statement: ...]*]

DEFINITION The IF statement allows a program to do a number of things based on the result of expression:

- 1.Branch to a line or label after the THEN if a condition is true; expression /=0
- 2.Execute statement(s) after the THEN if a condition is true; expression /=0
- 3.Branch to a line or label after the ELSE if a condition is false; expression=0
- 4.Execute statement(s) after the ELSE if a condition is false;expression=0

EXAMPLE

```
X=99
IF X=99 THEN PRINT"X=99":PRINT"HELLO: ELSE STOP
:
IF X=99 THEN "CHECK AGAIN"
END
:
"CHECK AGAIN"
IF X=100 THEN PRINT"YEP" ELSE PRINT"NOT TODAY!";:PRINT X
END
```

RUN

```
X=99
HELLO
NOT TODAY! 99
```

REMARK Complex strings will generate an error if used in an IF statement.

<i>Improper</i>	IF LEFT\$(A\$,2)="HI"THEN STOP
<i>Proper</i>	B\$=LEFT\$(A\$,2):IF B\$="HI" THEN STOP

See LONGIF,ELSE,XELSE,WHILE-WEND and DO-UNTIL for more ways of doing program comparisons.

Note: In many cases LONG IF is easier to read.



Also see SELECT CASE

INDEX\$ statement

FORMAT	INDEX\$ (<i>expression</i>) = <i>string expression</i> INDEX\$I (<i>expression</i>) = <i>string expression</i> INDEX\$D (<i>expression</i>)		
DEFINITION	INDEX\$ is a special array unique to ZBasic. Expression indicates an element number. <table border="0" style="width: 100%;"> <tr> <td style="width: 50%; vertical-align: top;"> <u>Statement</u> INDEX\$(n)=simple string INDEX\$I(n)=simple string INDEX\$D(n) </td> <td style="width: 50%; vertical-align: top;"> <u>Definition</u> Assigns a value to INDEX\$(n) Move element n (and all consecutive elements) up and INSERT simple string at INDEX\$ element n DELETE element n and move all consecutive elements down to fill the space. </td> </tr> </table>	<u>Statement</u> INDEX\$(n)=simple string INDEX\$I(n)=simple string INDEX\$D(n)	<u>Definition</u> Assigns a value to INDEX\$(n) Move element n (and all consecutive elements) up and INSERT simple string at INDEX\$ element n DELETE element n and move all consecutive elements down to fill the space.
<u>Statement</u> INDEX\$(n)=simple string INDEX\$I(n)=simple string INDEX\$D(n)	<u>Definition</u> Assigns a value to INDEX\$(n) Move element n (and all consecutive elements) up and INSERT simple string at INDEX\$ element n DELETE element n and move all consecutive elements down to fill the space.		
EXAMPLE	<pre> INDEX\$(0)="FRED" INDEX\$(1)="TOM" INDEX\$(2)="FRANK" : GOSUB"Print INDEX\$" INDEX\$I(1)="HARRY" GOSUB"Print INDEX\$" : INDEX\$D(0) GOSUB"Print INDEX\$" END : "Print INDEX\$": REM Routine prints contents of INDEX\$ FOR X=0 TO 4 PRINT X; INDEX\$(X) NEXT: PRINT RETURN </pre> <p>RUN</p> <pre> 0 FRED 1 TOM 2 FRANK 0 FRED 1 HARRY 2 TOM 3 FRANK 0 HARRY 1 TOM 2 FRANK </pre> <p style="margin-left: 200px;"><---Normal assignments</p> <p style="margin-left: 200px;"><---HARRY INSERTED between FRED and TOM</p> <p style="margin-left: 200px;"><---FRED is DELETED here</p> <p style="margin-left: 200px;"><--- Notice how values move from one element to another as items are inserted and deleted with INDEX\$I and D.</p>		
REMARK	INDEX\$ provides for memory efficient string array manipulation and lends itself very well to list management applications. See "Special INDEX\$ Array", INDEX\$ function, CLEAR, CLEAR INDEX\$ and MEM.		



Allows up to ten simultaneous INDEX\$ arrays. See INDEX\$ in your appendix.

FORMAT INDEXF (*string* [, *expression*])

DEFINITION INDEXF is a special INDEX\$ array function used to FIND a leading string within an INDEX\$ array quickly.

IF INDEX\$(1000) equaled "Hello", then X=INDEXF("Hel") would return 1000.

IF X=INDEXF("llo") X would equal -1 since "llo" would not be found. The leading characters are significant.

EXAMPLE

```

INDEX$(0)="FRED"
INDEX$(1)="MARY"
INDEX$(2)="TOM"
:
X=INDEXF("TOM")           <--- Search for TOM
PRINT X
:
PRINT INDEXF("MARY") <--- Search for MARY
:
PRINT INDEXF("RED")   <--- Search for RED
:
PRINT INDEXF("FRED",1) <--- Search for FRED starting at element 1

RUN

2      <----- TOM found at element two
1      <----- MARY found at element one
-1     <----- RED not found. The first characters are significant
-1     <----- FRED not found because search started at element 1
    
```

REMARK INDEX\$ provides for memory efficient string array manipulation and lends itself very well to list management and text editing applications.

See "Perpetual Sort" under "Special INDEX\$ Array". Also see INDEX\$,INDEX\$,INDEX\$,CLEAR,CLEAR INDEX\$ and MEM.



Allows up to ten simultaneous INDEX\$ arrays. See INDEX\$ in your appendix.

INKEY\$ function

FORMAT **INKEY\$**

DEFINITION INKEY\$ returns the character of the last key that was pressed or an empty string if no key was pressed.

EXAMPLE `WHILE A$<>"S": REM Press "S" to Stop
 DO
 A$=INKEY$
 UNTIL LEN(A$)
 A$=UCASE$(A$)
 PRINT A$;
 WEND
 END`

RUN

GHUIJD,KEUG FAQCCQ OPU...S <---When <S> is pressed program stops

```
REM An easy function you can use to get a key
LONG FN Waitkey$(local$)
  DO
    local$=INKEY$
  UNTIL LEN(Local$)
END FN=local$
:
key$=FN Waitkey$(key$)
PRINT key$
END
```

RUN

(user presses "b")

b

REMARK When using INKEY\$ for character entry, avoid having the TRON function active as this may cause pressed keys to be missed.

See INPUT,LINEINPUT,INPUT#,ASC and CHR\$. See your computer appendix for variations or enhancements.



Macintosh: See DIALOG (16) for way of doing INKEY\$ during event trapping.

MSDOS: INKEY\$ returns two characters for function keys. ON INKEY\$ does event checking for function keys. See appendix for specifics.

FORMAT **INP** (*expression*)

DEFINITION The INP function is used to read an input port. The function returns the value that is currently at the port specified by expression.

EXAMPLE `X=INP(1)`
 `PRINT X`
 `PRINT INP(G-1)`

RUN

0
255

REMARK Note: This function requires a knowledge of your computer hardware and may not be portable to other computers (may not be available on your version of ZBasic or may have an unrelated function).

See your computer appendix for specifics.



Not supported with this version. See INSLOT.



Not supported with this version. See OPEN"C" and "Toolbox" in the appendix for accessing hardware ports.

INPUT statement

FORMAT INPUT[(*@* or *%*)(*exprX*, *exprY*)][:][!][&*expr*,][*"string"*]; *var*[, *var* ...]

DEFINITION The INPUT statement is used to input values (string or numeric) from the keyboard into variables.

Multiple variables must be separated by commas (this is bad form since users often forget commas). If no value in INPUT, a zero or null string will be returned.

<i>@</i> (<i>xprX</i> , <i>exprY</i>)	Places cursor at text coordinate horiz,vert.
<i>%</i> (<i>exprX</i> , <i>exprY</i>) ;	Places cursor at graphic coordinate horiz,vert. Suppress carriage return/line feed.
!	Automatic Carriage return after maximum characters entered. User doesn't have to press <ENTER>.
& <i>expr</i> ,	Sets the maximum number of characters to be INPUT. Default is 255. Will not allow more than <i>expr</i> characters.
<i>"string"</i> ;	Optional user prompt will replace question mark. If a null string is used the question mark will be suppressed.
<i>var</i>	May be any variable type integer, single,double or string.

EXAMPLE See examples on following pages...

REMARK Differences in screen width may affect operation.

See LOCATE and PRINT for more information on cursor positioning. Also see INPUT#,LINEINPUT,LINEINPUT# and INKEY\$ for others ways of getting input.

See "Keyboard input" in the technical section.



Important Note: String lengths MUST be one greater than maximum INPUT length since a CHR\$(13) is temporarily added. Never define a string used in an INPUT or LINEINPUT as ONE.



In certain cases EDIT FIELD,MENU or BUTTON may be preferable. See appendix.

INPUT continued

EXAMPLES OF REGULAR INPUT

<u>EXAMPLE</u>	<u>RESULT</u>
INPUT A\$	Wait for input from the keyboard and store the input in A\$. Quotes, commas and control characters cannot be input. <ENTER> to finish. A carriage return is generated when input is finished (cursor moves to beginning of next line).
INPUT"NAME: ";A\$	Prints "NAME: " before input. A semi-colon must follow the last quote. A carriage return is generated after input (cursor moves to next line).
INPUT;A\$	Same as INPUT A\$ above, only the semi-colon directly after INPUT disables the carriage return (cursor stays on the same line).

EXAMPLES OF LIMITING THE NUMBER OF CHARACTERS WITH INPUT

<u>EXAMPLE</u>	<u>RESULT</u>
INPUT &10,A\$	Same as INPUT A\$ only a maximum of ten characters may be input. (&10) A carriage return is generated after input (cursor moves to the beginning of the next line). The limit of input is set for ALL variables, not each.
INPUT ;&3,I%	Same as INPUT &10, except the SEMI-COLON following INPUT stops the carriage return (cursor stays on line).
INPUT !&10,A\$	Same as INPUT & 10 except INPUT is terminated as soon as 10 characters are typed (or <ENTER> is pressed).
INPUT;!&10,"NAME: ",A\$	Same as INPUT ;&10,A\$ except no carriage return is generated (semi-colon). INPUT is terminated after 10 characters(&10 and Exclamation point). and the message "NAME: " is printed first.
LINEINPUT;!&5,"NAME: ";A\$	LINEINPUT A\$ until 5 characters or <ENTER> is pressed. (no carriage return after <ENTER> or after the 5 characters are input. Accepts commas and quotes.)

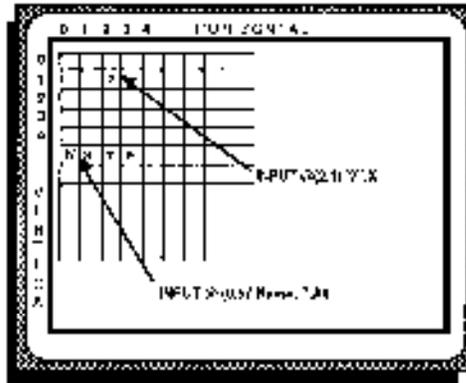
Note 1: Wherever INPUT is used, LINEINPUT may be substituted when commas, quotes or some other control characters need to be input (except with multiple variables).

Note 2: If more than one variable is INPUT, commas must be included from the user to separate input. If all the variables are not input, the value of those variables will be null.

INPUT statement

INPUT continued

INPUTTING FROM A SPECIFIC SCREEN LOCATION



`INPUT@(H,V);A$` Wait for input as TEXT screen POSITION defined by Horizontal and Vertical coordinates. No "?" is printed. A carriage return is generated.

`INPUT%(gH,gV);A$` Input from a graphic coordinate. Syntax is the same as "@". Very useful for maintaining portability without having to worry about different screen widths or character spacing.

`INPUT@(H,V);!10,"AMT: ";D#` Prints "AMT: " at screen position H characters over by V characters down. D# is input until 10 characters, or <ENTER> are typed in, and the input is terminated without generating a carriage return (the cursor DOES NOT go to the beginning of the next line).

`INPUT%(H,V);!10,"AMT: ";D#` Prints "AMT: " at Graphic position H positions over by V positions down. D# is input until 10 characters, or <ENTER>, are typed in, and input is terminated without generating a carriage return (the cursor DOES NOT go to the beginning of the next line).

Note: Replace INPUT with LINEINPUT whenever there is a need to input quotes, commas and control characters (except with multiple variables).

statement INPUT#

FORMAT INPUT # *expression, var[, var[, ...]]*

DEFINITION This statement will read INPUT from a disk or other device specified by expression until a carriage return, <COMMA>, End-Of-File or 255 characters are encountered.

Commas and leading spaces may be read into a string variable if the data on disk was enclosed in quotes, otherwise leading spaces and line feeds will be ignored.

See LINEINPUT# for ways of inputting commas, quotes and some control characters.

EXAMPLE

```
A$="HELLO"
B$="GOODBYE"
C$="WHAT?"
X#=12.345
:
OPEN"O",1"TEST.TXT":REM OPEN FOR OUTPUT
PRINT#1, A$, "B$", "C$", "X# <--- Quoted commas important with PRINT#
CLOSE#1
:
OPEN"I",1,"TEST.TXT":REM OPEN FOR INPUT
INPUT#1, X$,Y$,Z$,A# <--- INPUT# in same order and type as PRINT#
END
```

RUN

```
HELLO          GOODBYEWHAT?          12.345
```

REMARK See OPEN,CLOSE,PRINT#, and LINEINPUT#.

See your computer appendix for available devices.

Compatibility Note: ZBasic and MSBASIC have almost the same syntax with the following exceptions:

MSBASIC ALLOWS

```
PRINT#n, A$,B$,X#,C%
PRINT#n, A$ B$ C$
```

ZBasic REQUIRES

```
PRINT#n, A$,"B$","X#","C%"
PRINT#n, A$,"B$","C$"
```

If you remember that ZBasic puts the image to the disk just as if it were going to the printer or to the screen you will see why the syntax is important.

INSTR function

FORMAT **INSTR**(*expression*, *string1*, *string2*)

DEFINITION Finds the first occurrence of string 2 in string 1, starting the search at the position specified by expression.

expression Starting position of the search.
string1 String to be searched.
string2 String to search for.

EXAMPLE `Humble$="I am cool!"
PRINT INSTR(1,Humble$, "cool")
:
B$="am"
PRINT INSTR(1,Humble$, B$)
:
X=INSTR(1, Humble$, "FRED")
PRINT X
END`

RUN

```
6                    <---"Cool" started in the sixth position  
3                    <---"am" started at the third position  
0                    <---There was no "FRED" in the string.
```

```
Name$="Fred Smith"  
Lastname$=RIGHT$(Name$,LEN(Name$)-INSTR(1,Name$, " "))  
PRINT "Hello there Mr. ";Lastname$  
END
```

RUN

```
Hello there Mr. Smith
```

REMARK If the string is not found, zero (0) will be returned.

See LEFT\$,RIGHT\$,MID\$ and INDEXF.

FORMAT INT(*expression*)

DEFINITION Truncates all digits to the right of the decimal point of expression.

EXAMPLE

```

DEFDBL A-Z
DEFTAB 8
PRINT "      X", "ABS(X)", "INT(X)", "FRAC(X)", "SGN(X)"
:
FOR X = -15.0 TO +15.0 STEP 3.75
  PRINT USING"-##.###";X,
  PRINT USING"-##.###";ABS(X),
  PRINT USING"-##.###";INT(X),
  PRINT USING"-##.###";FRAC(X),
  PRINT USING"-##.###";SGN(X)
NEXT X
END
    
```

RUN

X	ABS(X)	<u>INT(X)</u>	FRAC(X)	SGN(X)
-15.00	15.00	-15.00	.00	-1.00
-11.25	11.25	-11.00	-.25	-1.00
- 3.75	3.75	-3.00	-.75	-1.00
.00	.00	.00	.00	.00
3.75	3.75	3.00	.75	1.00
7.50	7.50	7.00	.50	1.00
11.25	11.25	11.00	.25	1.00
15.00	15.00	15.00	.00	1.00

REMARK INT works the same as FIX in that expression will be restricted to the integer range of -32,768 to +32,767 only when the expression has not been defined as floating point.

INT is simply as a function that truncates an expression to a whole number.

To get the fractional part of a number use FRAC.

See FIX,SGN,ABS and FRAC.



INT range for the Macintosh is -2,147,483,648 to +2,147,483,647.

KILL statement

FORMAT **KILL** *simplestring*

DEFINITION KILL will erase a disk file specified by *simplestring*.

KILL functions either as a command or from within a program.

EXAMPLE

```
INPUT"File to erase:":A$
PRINT"Are you sure you want ";A$;" erased?";
INPUT B$
:
LONG IF B$<>"YES"
  PRINT"File not erased": STOP
XELSE
  KILL A$:PRINT A$;" is history."
END IF
:
END
```

RUN

```
File to erase: OldFile
Are you sure you want OldFile erased?
YES
Oldfile is history!
```

REMARK Use this statement with caution. When a file has been killed it is normally unrecoverable.

See RENAME,ERROR,ON ERROR,ERRMSG\$ and the "Files" section of this manual for more information.

This page intentionally left blank.

LEFT\$ function

FORMAT **LEFT\$** (*string*, *expression*)

DEFINITION LEFT\$ returns the left-most characters of string defined by expression. The string will not be altered.

EXAMPLE Quote\$="Early to Bed, Early to rise..."
 :
 PRINT LEFT\$(Quote\$, 5)
 :
 Part\$= LEFT\$(Quote\$, 12)
 PRINT Part\$
 :
 PRINT LEFT\$(Quote\$, 50);
 PRINT "Makes men healthy...at least"

RUN

```
Early
Early to Bed
Early to Bed, Early to rise... Makes men healthy...at least"
```

REMARK Also see RIGHT\$,MID\$,LEN,VAL,STR\$,INSTR,INDEX\$,SWAP and the "String Variable" section of this manual for more information about using strings.

FORMAT **LEN** (*string*)

DEFINITION Returns the number of characters that are stored in a string constant or string variable. If zero is returned it indicates a null (empty) string.

EXAMPLE

```
A$="FRED"  
B$="SMITH"  
:  
PRINT A$;" has";LEN(A$);" characters."  
PRINT B$;" has";LEN(B$);" characters."  
:  
PRINT LEN(A$)+LEN(B$)  
:  
PRINT LEN("Hello Fred")
```

RUN

```
FRED has 4 characters  
SMITH has 5 characters  
9  
10
```

REMARK The maximum length of a string is 255 characters. You may set the length of strings in ZBasic. See DIM,DEF LEN and the chapter on "String Variables" for more information about defining string length.

Since the first character of a string stored in memory is the length byte, PEEK(VARPTR(var\$)) will also return the length of a string.

The memory required for a string variable is the defined length + one for the length byte (256 bytes if not defined).

LET statement

FORMAT `[LET] variable = expression`

DEFINITION LET is an optional statement that may be used to assign an expression to a variable. Numbers, strings, numeric expressions, or other variables may be used to assign values to a variable if the types are compatible or convertible.

EXAMPLE `LET B=100`
 `PRINT B`
 `:`
 `LET B=B+10`
 `PRINT B`
 `:`
 `Z$="HELLO"+" THERE" <---Notice "LET" is optional`
 `PRINT Z$`

RUN

```
100
110
HELLO THERE
```

REMARK See SWAP, "Optimize expressions for Integer", "Math Expressions" and "Conversions Between Variable Types" for more information about assignments.

LINEINPUT statement

FORMAT **LINEINPUT**[(*@* or *%*)(*expr1*,*expr2*)][:][!][&*expr*,][*"string"*];*var*\$

DEFINITION The LINEINPUT statement is used to input characters from the keyboard into a string variable. It is different from INPUT in that quotes, commas and some control characters may also be entered. LINEINPUT is terminated when <ENTER> is pressed.

<i>@</i> (<i>expr1</i> , <i>expr2</i>)	Inputs from horizontal,vertical TEXT coordinate.
<i>%</i> (<i>expr1</i> , <i>expr2</i>)	Inputs from horizontal,vertical GRAPHIC coordinate.
;	Suppresses carriage-return/line-feed after input is complete. (disable inputs that cause scrolling or overwriting.)
!	Automatically executes a carriage return after the maximum number of characters are entered. The user doesn't have to press <ENTER>.
& <i>expr</i> ,	Sets the maximum number of characters to be input.
<i>"string"</i> ;	Optional string prompt will replace the question mark "?" normally shown with LINEINPUT.
<i>var</i> \$	Only string variables may be used with LINEINPUT.

EXAMPLE INPUT"Last name <COMMA> First name";A\$
PRINT A\$
:
LINEINPUT"Last name <comma> First name";B\$
PRINT B\$

RUN

Smith
Smith, Fred

REMARK See the chapter on "Keyboard Input" in the front of this manual for more examples.

The advantage of using LINEINPUT over INPUT is its ability to receive most of the ASCII character set except:

<ENTER>	CARRIAGE RETURN
<CTRL C>	CONTROL "C"
<BACKSPACE>	DELETE or LEFT ARROW
<CANCEL>	DELETE CURRENT LINE
<NULL>	NO CHARACTER



Important Note: String lengths MUST be at least one greater than the number of characters being input, otherwise a string overflow condition will destroy subsequent variables. Never use a one character string with LINEINPUT.

statement LINEINPUT#

FORMAT **LINEINPUT #** *expression ,variable\$*

DEFINITION This statement will input ASCII or TEXT data from a disk file specified by expression until <ENTER>, End-Of-File or 255 characters are encountered.

Useful for accepting commas, quotes and other characters that INPUT# will not accept. A good example of using LINEINPUT would be for reading an ASCII or TEXT file a line at a time (as in the example below).

EXAMPLE

```
REM Read a text file and print it to the screen
REM Routine compatible with all versions of ZBasic
:
ON ERROR GOSUB 65535: REM Error trapping on to check for EOF
:
OPEN"I",1,"TEXT.TXT"
:
Counter=0
:
WHILE ERROR=0: REM Read file until an EOF error
  LINEINPUT#1, A$
  PRINT A$
WEND
IF ERROR <> 257 THEN PRINT ERRMSG$(ERROR): STOP
ERROR=0
:
ON ERROR RETURN: REM Give error trapping back to ZBasic
END
```

REMARK The advantage of using LINEINPUT# over INPUT# is its ability to receive most of the ASCII character set. Leading linefeeds will be ignored on some systems.

If a CHR\$(0) or CHR\$(26) is encountered as a leading character it may assume EOF and set ERROR = End Of File (varies by computer).

Also see INPUT#,LINEINPUT and "Keyboard Input" in the front section of the manual.



These versions support an EOF function that would simplify the error trapping techniques used above. See the appropriate appendix for details about EOF:

```
OPEN"I",1,"TEXT.TXT"
Counter=0
:
WHILE EOF=0: REM Read until EOF
  LINEINPUT#1, A$
  PRINT A$
WEND:CLOSE#1
```

LIST command

FORMATS	[L]L[IST] [+][*] [L]L[IST] [+][*] <i>line or label</i> [L]L[IST] [+][*] <i>- line or label</i> [L]L[IST] [+][*] <i>line or label - line or label</i>	
DEFINITION	LIST (or L) is used from the Standard Line Editor to list the current program to the screen. LLIST will list the current program to a printer.	
	+ Suppress line numbers * Highlight keywords on the screen (some versions)	
EXAMPLE	<u>YOU TYPE</u> LIST or L LLIST LIST 100-200 LLIST-100 LIST "SUBROUTINE" LIST 100- or L100- <period> <UP ARROW> <DOWN ARROW> L+ LLIST+ L+-100 <SPACE> </> (slash key)	<u>ZBASIC RESPONDS</u> Lists complete program to the screen Lists complete program to the printer Lists lines from 100-200 Lists lines up to 100 to printer Lists the line with that label Lists the lines from 100 on Lists the last line listed or edited Lists previous line (or plus <+> key)* Lists next line (or minus <-> key)* Lists program without line numbers Lists to printer without line numbers Lists up to line 100 without line numbers PAUSE. <ENTER> continues PAGE AT A TIME: Lists 10 lines to the screen*

*See computer appendix for keyboard variations.

REMARK LIST automatically indents program lines two spaces between FOR-NEXT, DO-UNTIL, WHILE-WEND, LONG IF-XELSE-END IF and LONG FN-END FN structures.

See PAGE, WIDTH, WIDTH LPRINT and the chapter; "Formatting Listings".

Note: Labels may be used in place of line numbers.



LLIST+* will format listings to an Imagewriter or Laserwriter with no line numbers and with keywords in bold. While the output in of this format is extremely attractive and easy to read, it should be noted that listings will take about twice as long to print.

command LOAD

FORMATS **LOAD** ["] *filespec* ["]
LOAD * ["] *filespec* ["]

DEFINITION **LOAD** is used from the Standard Line Editor to load a ZBasic tokenized or a regular ASCII text file into memory.

ZBasic does not load tokenized files from other languages; the file must first be saved in TEXT or ASCII format.

If the program does not have line numbers they are added in increments of one.

LOAD* will strip away remarks and unnecessary spaces from an ASCII file releasing more room for the source and object code in systems with limited memory.

EXAMPLE **LOAD** **PROGRAM** <--- Loads a regular tokenized or text file
LOAD **"SOURCE"** <--- Double Quotes optional
LOAD* **THISONE** <--- Strips spaces and REM's while loading

REMARK Each operating system may require specific syntax for a drivespec.

Line numbers are optional in ASCII files.

If a program was created using another form of BASIC it must be in ASCII format before the ZBasic editor can load it.



These version of ZBasic support a Full Screen Editor that may support other forms of **LOAD**. See appropriate appendix for information about full Screen Editors.

LOC function

FORMAT LOC (expression)

DEFINITION Returns the byte pointer position within the current RECORD of the filenumber specified by expression.

EXAMPLE

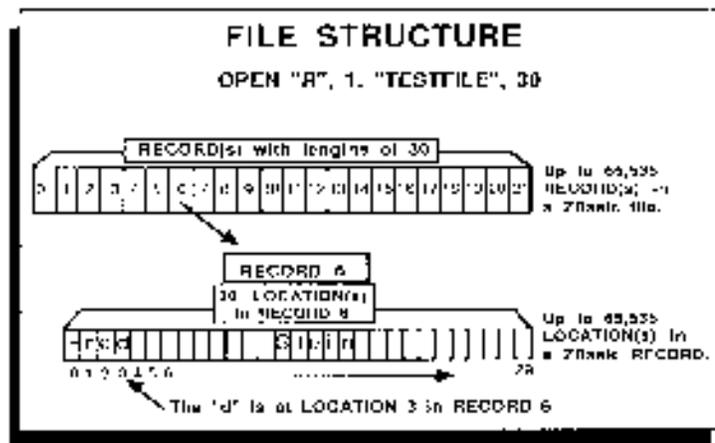
```

OPEN "R", 1, "TESTFILE", 30
RECORD#1, 6, 3 <---See illustration
PRINT LOC(1)
:
READ#1, Char$;1
PRINT LOC(1)
:
PRINT Char$
CLOSE#1
    
```

RUN

```

3
4
d
    
```



REMARK The LOC position is incremented to the next file position automatically when `READ#`, `WRITE#`, `INPUT#`, `LINEINPUT#` or `PRINT#` are used. `REC(filenumber)` returns the current RECORD. `LOF` returns the last record in the file. Also see "Files" section for more information.



The record length limits are different for these versions. See appendix.

statement LOCATE

FORMAT **LOCATE** *expr_x*,*expr_y*,[*expr_{cursor}*]

DEFINITION Positions the cursor to the coordinates given by *expr_y* and optionally turns on or off the cursor character (zero=off, not zero=on).

expr_x The horizontal coordinate (characters across)

expr_y The vertical coordinate (lines down)

expr_{cursor} Zero=cursor OFF. Non-zero = cursor ON

EXAMPLE `LOCATE 0,0` <---sets cursor in upper left corner
`LOCATE 10,0` <---sets Cursor 10 char to right at top
`LOCATE 0,10,0` <---sets Cursor 10th line down. Cursor OFF
`LOCATE 0,12,1` <---sets Cursor 12th line down. Cursor ON

REMARK This function is also useful with CLS LINE and CLS PAGE for clearing the screen to the end of line and end of page.

See "Screen and Printer Control", PRINT@, PRINT%, INPUT@, LINEINPUT@, LINEINPUT% and INPUT% for other ways of controlling the cursor positioning.

The ability to turn the cursor on or off may be limited by the hardware or software of some computers.



These versions of ZBasic allow you swap the horizontal and vertical coordinates under "Configure". This is handy for converting other BASIC programs that use the vertical coordinate first (not Apple DOS 3.3).

LOF function

FORMAT **LOF** (*expression*)

DEFINITION Returns the last valid RECORD number for the file specified by expression. LOF stands for Last-Of-File.



Important Note: This function may not return the last record correctly on some systems, especially if the record length of the file is different from the operating system's internal record length or if a file is opened with a different record length than that which it was opened originally. This is often remedied by simply setting the record length to the system default record length or the record length of which it was opened originally.

EXAMPLE See "Opening files for Append" in the "Files" section in the front of this manual for methods of getting a pointer to the last position in a file.

REMARK LOF returns the last record in the file. The default record length is 256 and may need to be changed to make LOF function properly.

See LOC and REC for getting file pointer information. See "Files" and "Disk Errors" for more information. Some systems return one for both record zero and record one.

Note to better usage: If you need to keep track of the last byte position of a sequential file or the last record of a random file, you might consider storing the last REC and LOC of a file in record zero before it is closed. Examples:

```
OPEN"O",1,"Textfile.txt"
RECORD#1,1 <---Set file pointer to record one (zero will store last REC and LOC)
PRINT#1,A$, "B$", "X", "Z# <---Save data
RECORD#1,0 <---- Position pointer to RECORD 0 to save last REC and LOC
R=REC(1):L=LOC(1)
WRITE#1, R,L <----Save pointers for future use
CLOSE#1
```

To add data to the end of the file later:

```
OPEN"R",1,"Textfile.txt"
RECORD#1,0
READ#1, R, L <--- Get last positions of file
RECORD#1, R,L <---- Position pointer to append data to the end of the file.
PRINT#1, A$ <---- Now you can append new data to the file
```

Don't forget to store the LOC and REC before closing! You could do the same thing with random files by saving the last record.



Also supports: LOF(*filenumber*,[*recordlength*]). LOF(1,1) would return the length of filenumber one in bytes.

FORMAT **LOG** (*expression*)

DEFINITION Returns the natural logarithm of expression (LN). LOG is the compliment of EXP.

Common LOG10= LOG(n)\LOG(10)

EXAMPLE PRINT LOG(2)
 X#=LOG(3)
 PRINT X#

RUN

.69314718056
1.09861228857

REMARK LOG is a scientific function. Scientific precision may be configured by the user differently from both single and double precision.

See "Configure" and "Math" in the beginning of this manual.

Also see COS,SIN,EXP,"^",ATN and TAN.

LONG FN statement

FORMAT **LONG FN** *name*[(*var*[, *var*[, ...]])]

```
.  
.   
END FN[= expression]
```

DEFINITION LONG FN is similar to DEF FN but allows the function to span over several lines. This is useful for your own functions that you can use with ZBasic.

A re-usable, non-line-numbered function may be saved to the disk with SAVE+ and retrieved later for use in other programs with APPEND.

The variables being passed to the function must not be arrays. The *expression* must be numeric for numeric functions and string for string functions.

EXAMPLE

```
LONG FN RemoveSpace$(x$)  
  WHILE ASC(RIGHT$(x$),1)=32  
    x$=LEFT$(x$,LEN(x$)-1)  
  WEND  
END FN= x$  
:  
Name$="ANDY            "  
:  
PRINT Name$;"*"  
:  
Name$=FN RemoveSpace$(Name$)  
PRINT Name$;"*"
```

RUN

```
ANDY            *  
ANDY*
```

```
REM Wait until key press. Return key in key$  
LONG FN WaitKey$(key$)  
  DO  
    key$=INKEY$  
  UNTIL LEN(key$)  
END FN=key$  
:  
Z$=FN WaitKey$(Z$)  
PRINT Z$
```

RUN

(returns key that was pressed)

REMARK Also see APPEND,SAVE+,DEF FN,FN statement,FN function and "Structure".

statement LONG IF

FORMAT **LONG IF** *expression*

```
.  
[XELSE]  
.   
ENDIF
```

DEFINITION LONG IF allows multiple line IF-THEN-ELSE structures. Very useful for breaking down complicated IF statements into more readable, logical structures. Two things happen based on the result of expression:

- * If expression is TRUE: Executes all the statements up to the XELSE (if used) and then exits at the END IF.
- * If expression is FALSE: Executes all the statements between the XELSE and END IF and then exits at the END IF. If XELSE is not used it will simply exit at the END IF.

EXAMPLE

```
INPUT "How old are you: ";Age%  
LONG IF Age% >=30  
    PRINT "You are Old aren't you !?"  
XELSE  
    PRINT "You're just a baby!"  
END IF
```

RUN

```
How old are you: 30  
You are Old aren't you!?
```

```
LONG IF Name$="Fred"  
    PRINT "Hello Fred...Long time no-see!"  
    PRINT "The balance you owe is";USING"$#####.##";Due#  
    PRINT "Thanks for asking."  
XELSE  
    PRINT "I don't know you! Go away!"  
END IF
```

RUN

```
Hello Fred...Long Time no-see!"  
The balance you owe is $1234.56  
Thanks for asking.
```

REMARK No loop may be executed within a LONG IF construct unless it is completely contained between a LONGIF and XELSE or between XELSE and ENDIF. The entire LONG IF construct must be completely contained within loops or nested loops in order to compile properly.

ZBasic will automatically indent program lines between LONG IF,XELSE and END IF two spaces. See the chapter about "Structure" for more information.

LPRINT statement

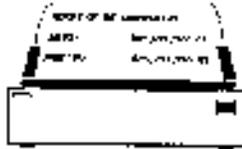
FORMAT LPRINT [*variables, constants,...*]

DEFINITION The LPRINT statement sends output to a printer.

To use LPRINT from the Standard Line Editor use a colon first (:LPRINT).

EXAMPLE

```
LPRINT "REPORT OF THE CORPORATION"  
LPRINT  
LPRINT  
LPRINT "SALES: ";TAB(50);USING"$##,###,###.##";Sales#(1)  
LPRINT  
LPRINT "PROFITS: ";TAB(50);USING"$##,###,###.##";Profits#(1)  
  
RUN
```



REMARK Some systems may lock up if a printer is not connected. See your hardware manual for required action.

See ROUTE 128,PRINT,LLIST,TAB,DEFTAB, PAGE, USING, WIDTH LPRINT and POS(1).



Macintosh: See DEF LPRINT,PRCANCEL,DEF PAGE,PRHANDLE, TEXT and ROUTE 128 in the appendix for more information about printing to the Imagerwriter and Laserwriter printers. See appendix for specifics.

MSDOS: To use more than one printer you may also use OPEN"I",1,"LPT2:" and use PRINT#1,[*variables,constants...*]. Be sure to close the printer device when finished. See MSDOS reference manual for more information about LPT2:,LPT1: and any other devices you may have available for your hardware.

Apple ProDOS and DOS 3.3: See DEF LPRINT for setting the printer slot.

statement MACHLG

FORMAT MACHLG{[bytes,...]} -or- {[words,...]} -or- {[variables],[,...]}

DEFINITION The MACHLG statement is used to insert bytes directly into a compiled program. These bytes may be machine language programs, variables or other items.

It may be used to insert machine language into memory without using POKE.

bytes Numbers from 0 to 255

words Numbers from 0 to 65535. They are stored in standard format

variables Will create the address where the variable is located. See appendix for specifics.

Note: ZBasic uses registers when calculating elements of an array variable. Contents of these registers may be destroyed.

EXAMPLE

```
X = LINE "Machine Language Routine"
FOR I = 0 TO 10
  PRINT PEEK(X+I);
NEXT I
END
:
"Machine Language Routine"
MACHLG 0,1,2,3,4,5,6,7,8,9,10
```

RUN

0 1 2 3 4 5 6 7 8 9 10

REMARK

See LINE,CALL,USR,DEFUSR,PEEK,POKE and the chapter about "Machine Language" in the technical section of this manual.



Important Note: Use of this statement requires knowledge of the machine language of the computer you are using. Machine language may not be portable to other computers.



Macintosh: Since the Macintosh is a 32 bit machine, MACHLG puts the code into word, not byte, positions.

MSDOS: See DEF SEG in appendix.

Apple ProDOS: See section entitled Machine Language Interface in appendix.

MAYBE function

FORMAT **MAYBE**

DEFINITION MAYBE is a random function that returns either a TRUE(-1) or FALSE(0) with equal probability.

MAYBE is faster than RND, convenient, and requires little program space.

EXAMPLE

```
DEFTAB = 8: DIM Coin$(1)
Coin$(0)="HEADS":Coin$(1)="TAILS"
:
"Flip a Coin"
DO
  X=X+1
  PRINT Coin$(MAYBE+1),
UNTIL X=25
END
```

RUN

```
HEADS HEADS TAILS HEADS TAILS
TAILS TAILS TAILS HEADS HEADS
TAILS TAILS HEADS TAILS TAILS
HEADS HEADS HEADS HEADS TAILS
HEADS TAILS TAILS TAILS HEADS
```

REMARK This function is useful anytime a 50% random factor is needed.

MAYBE with logical operators:

MAYBE	50% TRUE	50% FALSE
MAYBE AND MAYBE	25% TRUE	75% FALSE
MAYBE OR MAYBE	75% TRUE	25% FALSE

command MEM

FORMAT MEM[ORY]

DEFINITION Typing either MEM or MEMORY in command mode will return information about system memory use.

TEXT The number of bytes being used by the source code. The source code is that part of the program that you type in.

MEMORY The number of bytes remaining for program use (varies; see your computer appendix for details).

OBJECT The size of the object code after compiling.
Valid only immediately after RUN.

VARIABLES The number of bytes required for variables, INDEX\$ array, and disk I/O buffers. This varies dramatically by version. See computer appendix. *Valid only immediately after RUN.*

EXAMPLE MEM

```
00046 Text
41244 Memory
00039 Object
00388 Variable
(some versions may display more information)
```

REMARK These numbers are relative to that version of ZBasic being used. Varies significantly by computer.

See your computer appendix for more information.

Also see MEM function, CLEAR, CLEAR INDEX\$, CLEAR END, LOAD* and the chapter about "Converting Old Programs".

MEM function

FORMAT **MEM**

DEFINITION Returns the number of bytes available in the INDEX\$ array.

EXAMPLE `CLEAR 1000`
`PRINT MEM`
`A= MEM`
`INDEX$(0) = STRING$(49,"*")`
`PRINT MEM`

RUN

1000
950

REMARK See also INDEX\$, MEM command, and CLEAR INDEX\$. This function varies by version. See appendix for specifics.



MEM(index number) returns the memory available to that INDEX\$ (there are ten available on the Macintosh).

MEM(-1): Returns the maximum amount of memory available for variables. Also forces unloading of all unlocked memory segments. Returns a LongInteger.

INDEX\$ has many enhancements with this version. See appendix.



See appendix for various additions to the MEM function that return memory pointers to arrays, strings, BCD variables and more.

command MERGE

FORMATS **MERGE** ["] filespec ["]
 MERGE* ["] filespec ["]

DEFINITION MERGE is used to overlay a line numbered TEXT/ASCII program from disk onto the current program text in memory. Program being merged must be in ASCII (saved with SAVE*).

Incoming txt with the same line number(s) as resident text will replace resident text.

The asterisk is used to strip spaces and REM's from the incoming program.

EXAMPLE

```
010 REM Program one
120 DO
130  I$=INKEY$
140 UNTIL LEN(I$)
SAVE* "PROG1"
NEW

10 REM Program two
20 PRINT "MAIN MENU"
30 PRINT
40 PRINT "1. Do Inventory"
50 PRINT "2. Print Inventory"
60 PRINT "3. Delete Inventory"
MERGE "PROG1"
LIST

00010 REM Program one            <---- Line from first program overwrote this line
00020 PRINT "MAIN MENU"
00030 PRINT
00040 PRINT "1. Do Inventory"
00050 PRINT "2. Print Inventory"
00060 PRINT "3. Delete Inventory"
00120 DO                            <---First program merged here
00130  I$=INKEY$
00140 UNTIL LEN(I$)
```

REMARK MERGE has the same affect as manually typing in text.

Programs that were written in another BASIC must be in ASCII format before being MERGED into ZBasic.

Also see LOAD,SAVE*,RENUM,APPEND and DELETE

MID\$ function

FORMAT MID\$ (*string* , *expr1* [, *expr2*])

DEFINITION Returns the contents of string starting at position *expr1*, and *expr2* characters long.

string The string from which the copy will occur.

expr1 The distance from the left that the copy will begin.

expr2 Optional parameter that determines how many characters will be copied. If omitted, all characters from *expr1* to the end of the string will be copied.

EXAMPLE

```
A$="The Sun Shines Bright"  
:  
PRINT MID$(A$,5,3)  
:  
Z$=MID$(A$,15)  
PRINT Z$  
:  
FOR Pointer = 1 TO LEN(A$)  
  PRINT MID$(A$,Pointer,1)  
NEXT
```

RUN

```
Sun  
Bright  
T  
h  
e
```

```
S  
u  
n  
.  
.  
.
```

```
INPUT"First and Last name please:";Name$  
PRINT "Thank you Mr. ";MID$(Name$,INSTR(1,Name$,"")+1)
```

RUN

```
First and Last name please: Fred Smith  
Thank you Mr. Smith
```

REMARK See LEFT\$,RIGHT\$,INSTR,LEN, STR\$ and the MID\$ statement.

FORMAT MID\$ (*string1*, *expr1* [, *expr2*]) = *string2*

DEFINITION Replace a portion of *string1* starting at *expr1*, with *expr2* characters of *string2*.

string1 Target string. *String2* will be inserted or layed over this string.

string2 String to be inserted or layed over *string 1*.

expr1 Distance from the left of *string1* where overlay is to begin

expr2 How many characters of *string2* to insert into *string1*. Using 255 will assure that all characters are used.

EXAMPLE

```
A$ = "SILLY BOY"  
B$ = "SMART"  
:  
MID$(A$,1,5) = B$  
PRINT A$
```

RUN

```
SILLY BOY  
SMART BOY
```

REMARK This function is very useful for altering selected portions of strings.

Also see RIGHT\$,LEFT\$,MID\$ function,STR\$,INSTR,VAL,LEN,SPACE\$,STRING\$.

MKB\$ function

FORMAT MKB\$ (*expression*)

DEFINITION Returns a string which contains the compressed floating point value of a ZBasic BCD expression.

This function works with either single or double precision. The amount of string space used will vary depending on the digits of precision configured. See "Configure".

To return the floating point values stored in strings use the CVB function.

EXAMPLE

```
A$=MKB$(991721.645643)
PRINT "The length of A$=";LEN(A$)
X!=CVB(A$)
PRINT X!
:
PRINT
:
B$=MKB$(991721.645643)
PRINT "The length of B$=";LEN(B)
X#=CVB(B$)
PRINT X#
```

RUN

The length of A\$=4 <--- Value returned depends on configured precision
991722

The length of B\$=8 <--- Value returned depends on configured precision
991721.645643

REMARK Since ZBasic automatically compresses and decompresses BCD variables when using READ# and WRITE#, this function is of primary interest to those people that need to conserve memory for other reasons.

See also CVB,CVI,READ#,WRITE# and MKI\$.

See your appendix for default accuracy and variations.

FORMAT MKI\$ (*expression*)

DEFINITION Returns a two character string which contains a two byte integer specified by expression.

To extract the integer stored in a string with MKI\$, use the CVI function.

EXAMPLE

```
A$=MKI$(12345)
PRINT"Length of A$=";LEN(A$)
B%=CVI(A$)
PRINT B%
PRINT
:
A$=STR$(12345)
PRINT "Length of A$=";LEN(A$)
PRINT VAL(A$)
```

RUN

```
Length of A$=2
12345          <--- MKI$ saves space...(4 bytes compared to below)

Length of A$=6
12345          <--- Leading blank reserved for the "SIGN"
```

REMARK Used in older versions of BASIC to convert integers to strings for FIELD statements. ZBasic does this automatically when using READ# and WRITE#. Nevertheless, MKI\$ and CVI are still useful for packing strings to save memory-- especially on systems with limited memory.

See also CVI,CVB,READ#, WRITE# and MKB\$.



Use DEFSTR LONG to allow MKI\$,CVI,HEX\$,OCT\$ and BIN\$ to work with LongIntegers. Use DEFSTR WORD to set back to regular integer. Note that MKI\$ returns a four byte string with LongIntegers.

MOD operator

FORMAT *expression1* **MOD** *expression2*

DEFINITION MOD returns the remainder of an integer division with the sign of expression1.

EXAMPLE PRINT "9 DIVIDED BY 2=";INT(9/2);"REMAINDER=";9 MOD 2

RUN

9 DIVIDED BY 2= 4 REMAINDER= 1

PRINT "-4 DIVIDED BY 2=";INT(-4/2);"REMAINDER=";-4 MOD 2

RUN

-4 DIVIDED BY 2= -2 REMAINDER= 0

REMARK MOD replaces the old BASIC routines for finding the remainder of a division and is also much faster:

OLD BASIC: $X = (X - \text{INT}(X/N) * N)$

ZBasic: $X = X \text{ MOD } N$

FORMAT MODE expression

DEFINITION MODE is used to set the screen graphics or text format.

Most computers offer a number of different character and/or graphic modes. Use MODE to choose the mode most applicable to the program.

For most systems EVEN modes are character graphics and ODD modes are regular graphics. Not all machines have graphic capability. MODE for some popular microcomputers:

Mode number	MSDOS type		APPLE IIe, IIc		TRS-80 I, III	
	Text	Graphic	Text	Graphic	Text	Graphic
0	40x25	character	40x24	character	32x16	character
1	40x25	40x40	40x24	40x40	32x16	40x40
2	80x25	character	80x24	character	32x16	character
3	80x25	80x40	80x24	80x40	32x16	80x40
4	80x25	40x40	80x24	character	32x16	character
5	80x25	80x40	80x24	80x40	32x16	80x40
6	80x25	character	80x24	character	32x16	character
7	80x25	80x40	80x24	80x40	32x16	80x40
8	40x25	40x40	40x24	character	32x16	40x40
9	40x25	40x40	40x24	40x40	32x16	40x40
10	80x25	character	80x24	character	32x16	character
11	80x25	80x40	80x24	80x40	32x16	80x40
12	80x25	character	80x24	character	32x16	character
13	40x25	80x40	40x24	80x40	32x16	80x40
14	80x25	character	80x24	character	32x16	character
15	80x25	80x40	80x24	80x40	32x16	80x40

MACINTOSH		CP/M-80	
Text	Graphic	Text	Graphic
Mode 0 for raster and character	ALL Macintosh APPLES	80x24	80x40 APPLES

Be sure to read the appropriate appendix for exact mode designations.

REMARK MODE will reset COLOR to the default, usually the darkest background and lightest foreground, and may clear the screen with some systems.



Macintosh: MODE is ignored with the Macintosh. See the TEXT statement for setting character styles and sizes. To emulate other computers you will probably want to use Monaco or Courier mono-spaced fonts. TEXT font, size, face, mode.

MSDOS: Modes 16-19 support EGA modes. Mode 20 supports Hercules graphics. See appendix for details.

MOUSE function

FORMAT **MOUSE** (*expression*)

DEFINITION Returns information concerning the position and status of a MOUSE or JOYSTICK if one is connected to the system. The following values are returned.

MOUSE(0) Initializes the MOUSE on some systems (initialization is required on the Apple // ProDOS and DOS 3.3 versions).

MOUSE(1) Returns the horizontal coordinate of the mouse.

MOUSE(2) Returns the vertical coordinate of the mouse.

MOUSE(3) Returns 0 if button not pressed. Non-zero if button pressed.

EXAMPLE

```
MODE 5 :REM GRAPHIC MODE
CLS
X= MOUSE (0)                      <---Initialize mouse
:
WHILE LEN(INKEY$)=0               <--- Press any key to stop
  LONG IF MOUSE (3)               <--- If button down then ok to draw
    PLOT MOUSE (1), MOUSE (2) <--- Plot where mouse (or joystick) is.
  END IF
WEND
```

REMARK

The above example uses a mouse to draw on the screen. A joystick may also be used (depending on the system). See your computer appendix for hardware device specifics that may apply to these functions.

Also see DEF MOUSE.



Macintosh Note: You may use the mouse functions above or configure ZBasic for MSBASIC Mouse compatibility using DEF MOUSE=1. See Mac Appendix.

MSDOS: Compatible with Microsoft Mouse. ZBasic has to be configured to support a mouse. See "Configure" in MSDOS appendix. If MOUSE(0) <> 0 then a mouse is installed. MOUSE(3) return 0-3; Zero if both buttons up, three if both buttons down, one or two if one button is pressed. MOUSE(4) and MOUSE(5) hide and show the mouse cursor. DEF MOUSE=0 for Mouse, 1 or 2 for joysticks, 3 for lightpens.

Apple ProDOS and DOS 3.3: Compatible with AppleMouse or joysticks. Use DEF MOUSE=0 for AppleMouse or DEF MOUSE=1 for Joysticks. If using a joystick MOUSE(3) returns 0-3. Zero if both buttons up, three if both buttons down, one or two if one button pressed. See appendix for specifics.

Z80: MOUSE IS NOT SUPPORTED with Z80 versions of ZBasic.

statement NAME



FORMAT **NAME** *string1* **AS** *string2*

DEFINITION Renames a file with a filename of *string1* to *string2*. Same as the RENAME statement except for syntax. This statement is provided to make ZBasic compatible with other BASIC languages.

EXAMPLE **DIR**

```
FRED.BAS                    TOM.BAS
DICK.BAS                    HARRY.BAS
```

```
NAME FRED.BAS AS GEORGE.BAS
```

DIR

```
GEORGE.BAS                 TOM.BAS
DICK.BAS                    HARRY.BAS
```

REMARK See RENAME for more information.



Not available on Apple // or Z80 versions of ZBasic. See RENAME.

NEW command

FORMAT **NEW**

DEFINITION **NEW** is used to clear the text buffer of the current program.

Since programs that have been erased in this manner are impossible to recover, **SAVE** your program first!

EXAMPLE **LIST+**

```
CLS
PRINT"THIS IS A PROGRAM  ";
PRINT"WHICH IS ABOUT TO BE LOST FOREVER AND EVER..."
END
```

```
NEW
LIST
```

(Nothing listed...)

REMARK Use this command with care. See **LOAD**.

statement NEXT

FORMAT FOR var = *expression1* TO *expression2* [STEP *expression3*]
.
.
NEXT [*variable* ,[*variable* ...]]

DEFINITION The NEXT statement is used as the end marker of a FOR loop. There must be a matching NEXT for every FOR, otherwise a Structure Error will occur at compile time.

EXAMPLE

```
FOR Count1= 1 TO 2
  FOR Count2 = 2 TO 4 STEP 2
    PRINT Count1, Count2
  NEXT Count2, Count1
```

RUN

```
1          2
1          4
2          2
2          4
```

```
FOR X= 1 TO 2
  FOR Y= 1 TO 2
    PRINT X,Y
  NEXT
NEXT
```

RUN

```
1          1
1          2
2          1
2          2
```

REMARK The variable(s) following the NEXT statement are optional; however, if used they must match the corresponding FOR variable(s).

A FOR-NEXT loop will execute AT LEAST ONCE!

A Structure Error will specify the line number if there is an extra NEXT, or will specify line 65535 if a NEXT is missing. ZBasic automatically indents all loop structures when you LIST your program. This may be used to find where the missing NEXT is located by simply following the program listing back to the point where the extra indent ends.

See "Loops" in the front of this manual and; WHILE-WEND, DO-UNTIL, LONGIF-ELSE-ENDIF for other loop and structure types.

NOT operator

FORMAT NOT *expression*

DEFINITION NOT returns the opposite of expression. True is False, False if TRUE. This is equivalent to changing a logical true (-1) to a logical false(0) and vice versa.

With Boolean (binary) operations, the NOT function will toggle all bits in expression. That is, all bits that are one will be changed to zero, and all bits that are zero will be changed to one.

EXAMPLE A\$="Hello"
IF NOT A\$="Bye" THEN PRINT"True, it is False"
END

RUN

True, it is False

REMARK A logical true is -1 and logical false is 0. Also see XOR,OR,AND.

NOT condition TRUE(-1) if condition FALSE, else FALSE(0) if TRUE

NOT

NOT 1 = 0
NOT 0 = 1

BOOLEAN "16 BIT" LOGIC

NOT 11001100 NOT 01111011
= 00110011 = 10000100



Will also function with 32 bit LongIntegers.

FORMAT **OCT\$** (*expression*)

DEFINITION OCT\$ returns a 6 character string which represents the Octal value (base 8) of the result of expression truncated to an integer. Octal digits are from 0-7.

<u>OCTAL</u>	<u>DECIMAL equivalent</u>
0-7	0-7
10	8
11	9
12	10
13	11
14	12
15	13
16	14
17	15
20	16

EXAMPLE The following program will convert a decimal number to Octal or an Octal number to decimal:

```
CLS
DO
  INPUT "Decimal number: ";Decimal%
  PRINT "Octal Equivalent: ";OCT$(Decimal%)
  :
  INPUT "Octal number: ";Octal$
  Octal$="%O"+Octal$
  PRINT "Decimal Equivalent: ";VAL(Octal$)
UNTIL (DECIMAL%=0) OR (LEN(Octal$)=2)
```

RUN

```
Decimal number: 8
Octal Equivalent: 000010
```

```
Octal number: 100
Decimal Equivalent: 80
```

REMARK Conversions are possible from any base to any other base that ZBasic supports.

See the Chapter "Numeric Conversions" in the front of this manual. See also BIN\$, HEX\$ and UNS\$.



Use DEFSTR LONG if you want to use OCT\$,HEX\$,BIN\$,UNS\$,MKI\$ or CVI with LongIntegers. Use DEFSTR WORD to set back to regular integer.

ON ERROR statement

FORMAT	ON ERROR GOSUB <i>Line or label</i> ON ERROR <i>Return</i> ON ERROR GOSUB 65535
DEFINITION	The ON ERROR allows the user to enable and disable disk error trapping. If ON ERROR is not used ZBasic will display disk errors as they occur and give the user the option of continuing or stopping. Options offered with ON ERROR: ON ERROR GOSUB 65535 Enable user disk error trapping. Errors are returned using the ERROR function. You must check for errors---ZBasic will not when this parameter is set. ON ERROR GOSUB <i>line</i> If a disk error occurs the program does a GOSUB to the line or label specified. ON ERROR RETURN Disable user disk error trapping. ZBasic will trap the disk errors and give error messages at runtime.

EXAMPLE

```
ON ERROR GOSUB 65535: REM Enable disk error trapping
"Start"
OPEN "I" ,1, "TEST"
IF ERROR GOSUB "Disk error"
GOTO "Start"
program continues...
:
:
"Disk error"
LONG IF (ERROR AND 255)=3: REM Check for File not found error
PRINT"Check that correct diskette is in drive: <ENTER>";
DO
UNTIL LEN(INKEY$)
ERROR=0:RETURN
XELSE
PRINT"A Disk Error has occured:";ERRMSG$(ERROR)
PRINT"<C>ontinue or <S>top?";
DO
temp$=UCASE$(INKEY$)
UNTIL (temp$="C") OR (temp$="S")
IF temp$="C" THEN ERROR=0: RETURN
END IF
PRINT"Program aborted!"
ERROR=0
STOP
```

REMARK Also see ERROR and ERRMSG\$ and the chapter about "Disk Error Trapping" in the "Files" section of the manual.
See RETURN line for another way of returning from ON ERROR GOSUB line.



Important Note: Always remember to set ERROR=0 after a disk error occurs when you are doing the disk error trapping. Failure to do this will cause ZBasic to continue to return a disk error condition.

statement ON GOSUB

FORMAT **ON** *expression* **GOSUB** *line* [, *line* [, *line*...]]

DEFINITION The ON GOSUB statement is used to call one of several subroutines depending on the value of expression.

The ON statement will call the first subroutine if the expression evaluates to one, to the third subroutine if the expression evaluates to three and so on.

The RETURN statement at the end of a subroutine will return the program to the statement immediately following the ON GOSUB.

EXAMPLE

```
"Inventory Menu"
CLS
PRINT "1. Inventory"
PRINT "2. Print Listing"
PRINT "3. Month End"
PRINT "4. EXIT"
PRINT
PRINT "Enter item wanted: ";
:
DO
  Item%=VAL(INKEY$)
UNTIL (Item% >0) AND (Item% <5)
:
ON Item% GOSUB "Inventory", "Print", "EOM", "Exit"
GOTO "Inventory Menu"
END
:
"Inventory"
RETURN
:
"Print"
RETURN
:
"EOM"
RETURN
:
"Exit"
END
```

REMARK ZBasic will truncate expression to an integer. For example, if expression equaled 1.9, the ON statement would go to the first line (INT(1.9)=1).

If expression <=0 or > (number of line numbers listed), the program will continue on to the next statement in the program.

ON GOTO statement

FORMAT **ON** *expression* **GOTO** *line* [, *line* [, *line*...]]

DEFINITION The ON GOTO statement is used to branch, or jump, to one of several portions of a program depending on the value of expression.

The ON statement will jump to the first subroutine if the expression evaluates to one, to the third subroutine if the expression evaluates to three, and so on.

EXAMPLE

```
A=RND(4)
ON A GOTO "ONE", "TWO", "THREE", "Last"
END
:
"ONE"
PRINT 1
END
:
"TWO"
PRINT 2
END
:
"THREE"
PRINT 3
END
:
"Last"
PRINT 4
END
```

RUN

4

REMARK ZBasic will truncate expression to an integer. For example, if expression equaled 1.9, the ON statement would go to the first routine (INT(1.9)=1).

If expression ≤ 0 or $>$ (number of line numbers listed), the program will continue on to the next statement in the program.

See "Structure".

statement OPEN

FORMAT **OPEN "I",** [#] *filename*, *filename* [, *record length*]
 OPEN "O", [#] *filename*, *filename* [, *record length*]
 OPEN "R", [#] *filename*, *filename* [, *record length*]

DEFINITION The OPEN statement is used to access a data file. Once a file is opened, information may be read from or written to the file depending on the way the file was opened. The first argument determines access:

"R" Read/write file: Open file if it exists, create the file if it doesn't.

"I" Read only file: Open file for input. If file doesn't exist, a disk error occurs (file not found error).

"O" Write only file: Open file for output. Overwrites the old file.

filename The number you assign to a file which is subsequently used with file commands like READ#,WRITE#,INPUT#,LINEINPUT#,PRINT#,REC,LOC and LOF.

filename The filename as it appears in a directory. See your DOS manual and the appendix in this manual for information about drive specifiers, pathnames, sub-directories or whatever syntax is used for that computer.

record length Optional record length to be used with that file (default is 256).

EXAMPLE REM Open a file for READ and WRITE
 OPEN "R",1,"INVEN", 180
 :
 REM Open a file for Input only
 OPEN "I", File%, D\$+"INVEN", 180
 :
 REM Open a file for Output only
 OPEN "O",2, Filename\$

REMARK To configure ZBasic to have more than two files open at a time; see "Configure". Each file buffer will require between 160 and 1024 bytes of memory depending on the Disk Operating System and your version of ZBasic. No more than 99 files may be open at one time.

See your computer appendix for more information about file types, changing directories and more. Also see INPUT#,PRINT#,READ#,WRITE#,LOC and REC.



TO INSURE DATA INTEGRITY, ALWAYS CLOSE OPEN FILES BEFORE EXITING YOUR PROGRAM.

continued...

OPEN statement

OPEN continued



Macintosh: Extra parameters included:

volume%

The number you get from FILE\$\$ that sets the folder or root location of the file. Much easier than pathname specifiers. See appendix for details. Also see FILE\$, EJECT, EOF, LOF, "File size", APPEND and pathnames. Example of volume number:

```
OPEN "type", fnum, "filename", 200, volume%
```

Additional types

"R[R]", "O[R]", "I[R]", "A[R]" and "R[D]", "O[D]", "I[D]", "A[D]"
The optional "R" or "D" after the file type specifies opening the resource fork (R) or data fork (D). The data fork is the default. See appendix for specifics. The "A" type opens a file for append. Also see APPEND for positioning the file pointer to the end.

Pathnames

Pathnames are supported like: Root:Folder:Fred



MSDOS: There are many ways to specify, create or remove directories and sub-directories. See PATH\$, CHDIR, MKDIR and RMDIR in the appendix.



Apple ProDOS: See PATH. Filenames may contain pathname information like: PROFILE/ZBASIC/SOURCE. See appendix for details.

Apple DOS 3.3 uses CP/M type drivespecs like: A: instead of D1, B: instead of D2, etc. Filetype is specified by a leading exclamation mark and a number:

OPEN "-", filename, "[!type][drivespec] filename", record length

!type=

1= Text file

5= S type file

2= Integer BASIC

6= Relocatable file type

3= Applesoft BASIC

7= A type file

4= Binary file

8= B type file

Example: OPEN "-", fnum, "!4 A:FRED", 200



CP/M-80: You may use a drive specifier in the filename:

```
OPEN "-", n, "A:Fred.DAT", 200
```

TRS-80: You may use a drive specifier in the filename:

```
OPEN "-", n, "Fred/DAT.password:1", 200
```

statement OPEN "C"

FORMAT OPEN "C",-1 or -2,[*baud rate*],[*parity*],[*stopbit*],[*word length*]]]

DEFINITION This statement is used to set serial communication port parameters. If any of the parameters are omitted the default will be used.

-1	Serial port one	
-2	Serial port two	
baud rate	110, 150, 300(default), 600, 1200, 2400, 4800, 9600	
parity	0 = none<-- default 1 = odd 2 = even	
stopbit	0 = one <-- default 1 = two	
word length	0 = 7 bits 1 = 8 bits	<-- default

EXAMPLE

```

REM A Very Cheap Terminal Program
OPEN"C",-1, 300 <---Change parameters as needed
DO
  READ#-1, A$;0 <---(;0) Won't "Hang" if nothing at port
  IF LEN(A$) THEN PRINT A$;
  :
  A$=INKEY$
  IF LEN(A$) THEN PRINT#-1,A$;
UNTIL A$=" ]" <--- Set a key to stop

```

REMARK Serial ports may be accessed using the same statements used in disk I/O: PRINT# INPUT#,LINE INPUT#,READ#, and WRITE#. In all of these statements, the port is not read or written to until the status indicates that the port is ready.

The one exception to the paragraph above is when READ# is used to read a string of zero length. In this case, the character will be returned if ready, otherwise a null string will be returned (similar to the INKEY\$ function) (Not supported with CP/M).

A port does not have to be opened in order to be accessed. The OPEN "C" statement is used only to set the current port parameter values. Without this statement, the port will simply use the parameters to which it was last set.



All versions have a number of machine specific parameters. See appendix for important details.

continued...

OPEN "C" statement

OPEN "C" continued

The following are examples of sending or receiving files over a modem or serial line. Check appendix and hardware manuals for specifications.

Add your own line numbers, and modify programs as needed. Save with SAVE+ to use later.

SEND FILES TO ANOTHER COMPUTER

```
"SEND FILES"
LINEINPUT"File to send: ";File$
IF LEN(File$)=0 THEN STOP: REM No file? STOP
:
OPEN"I",1,File$
ON ERROR GOSUB 65535: REM Catch errors
:
OPEN"C",-1,300: REM Change parameters as needed
:
DO
  LINEINPUT#1, Line$
  IF LEN(Line$) THEN PRINT#-1, Line$
  DO <---- This DO loop is an example of "Handshaking" remove
    READ#-1,A$;0 this loop, and the PRINT# below, if not needed.
  UNTIL ASC(A$)=1
UNTIL ERROR
:
IF ERROR=0
CLOSE#1
PRINT#-1,"*END*": REM Tell receiver "All Done!"
RETURN
```

RECEIVE FILES FROM ANOTHER COMPUTER

```
"RECEIVE FILES"
LINEINPUT"Filename to Receive: ";File$
IF LEN(File$)=0 THEN STOP: REM No File? STOP
:
OPEN"O",1,File$
:
OPEN"C",-1,300: REM Change parameters as needed
:
DO
  LINEINPUT#-1, Line$
  IF Line$<>"*END*" THEN PRINT #1, Line$
  PRINT#-1, CHR$(1); <--- Goes with "Handshaking" DO Loop above.
UNTIL (Line$="*END*")
:
CLOSE#1
RETURN
```

FORMAT *expression OR expression*

DEFINITION Performs a logical OR on the two expressions for IF THEN testing and BINARY operations. If either or both conditions are true the statement is true. See truth table below.

In binary/boolean operations if either bit is one than a one is returned.

EXAMPLE A\$="HELLO"
IF A\$="GOODBYE" OR A\$="HELLO" THEN PRINT"YES"

RUN

YES

REMARK Truth table for the OR function.

condition OR condition TRUE(-1) if either or both is TRUE, else FALSE(0)

<u>OR</u>	<u>BOOLEAN "16 BIT" LOGIC</u>			
1 OR 1 = 1		00000001		10000101
0 OR 1 = 1	OR	00001111	OR	10000111
1 OR 0 = 1	=	00001111	=	10000111
0 OR 0 = 0				

Also see AND,XOR and NOT.



Functions with 32 bit LongInteger as well.

OUT statement

FORMAT **OUT** *port,data*

DEFINITION The OUT statement sends data to the specified port number.

EXAMPLE OUT 1,12
 :
 A=6:B=9
 OUT A,B
 :
 OUT A/2,B/3
 END

REMARK This statement is microprocessor dependent and works only with Z80 and 8086 type processors.

Also see INP for a way of reading data in from the port.



Not supported with these versions.

FORMAT PAGE

DEFINITION Returns the current line position of the printer. The first line is line zero.

EXAMPLE PAGE <---Also see PAGE statement
PRINT PAGE
LPRINT
LPRINT
LPRINT
PRINT PAGE

RUN

0
3

REMARK This function is similar to POS except the line position is returned instead of the character position.



Important Note: If your operating system uses forms control and checks lines per page, you must disable the operating systems forms control or ZBasic's PAGE.



See CSRLN in the MSDOS appendix for getting the line position of the screen cursor.

PAGE function

FORMATS **PAGE** [[*expression1*][,[*expression2*]],[*expression3*]]]

DEFINITION **PAGE** is used to format output to the printer and to control the number of actual lines per page, printed lines per page and top margin. Following is a description of the parameters:

PAGE	Without parameters will send a page feed to the printer. this forces the print head to move to the defined position of the top of the next page.
<i>expression1</i>	The number of printed lines per PAGE
<i>expression2</i>	The number of actual lines per PAGE. Also resets the count to zero (normally 66 lines per page).
<i>expression3</i>	Lines for the top margin. This number is a subset of <i>expression1</i> . If the line count is zero, this many linefeeds will be output immediately.

EXAMPLE **PAGE** 60,66,3 <--- Sets Listings to 60 lines per page with 3 lines as top margin. Skips perforations nicely.

REMARK WIDTH LPRINT should be set to your printer's character width for proper PAGE operation when doing LLIST.

See PAGE function.

To disable PAGE use PAGE 0



Important Note: If your operating systems uses forms control and checks lines per page, you must disable the operating systems forms control or ZBasic's PAGE.



FORMATS **PATH**

DEFINITION PATH or PATH type commands are available on many versions of ZBasic that support multi-level directories. Rather than give the exact syntax for each machine this page gives a general overview. See your computer appendix for specifics.

MSDOS See PATH\$ function in the appendix. This allows you to get the current path name so that you can return to that sub-directory. Syntax is PATH\$(drive number). Note: Drive A=1, B=2, ...

Pathname syntax example: C:\ZBasic\TEMP

Apple ProDOS See PATH command in the appendix. Also see the example function on the master disk called: PREFIX.SAMPLE for ways of getting ProDOS pathnames during runtime.

Pathname syntax example: /PROFILE/ZBASIC/OBJECT

Pathnames not supported with DOS 3.3 version.

Macintosh The most appropriate way of specifying where a file is located is using the volume number. This is recommended in "Inside Macintosh". Volume numbers are obtained easily using the FILES\$ function. See Macintosh appendix.

Nevertheless, pathnames are supported and may be used.

Pathname syntax example: Fred:Tom:Harry

Z80 Pathnames are not supported since the operating systems for this CPU do not currently implement sub directories.

EXAMPLE See your appendix for examples.

REMARK This command varies significantly by computer type.

See DIR,OPEN and also be sure to see your appendix for specifics.



Pathnames are not supported with Apple DOS 3.3 or Z80 versions of ZBasic.

PEEK function

FORMAT **PEEK [WORD]** (*expression*)
PEEK LONG (*expression*)*

DEFINITION Returns the contents of the memory location(s) specified by expression:

PEEK	Returns a one byte number (0-255)
PEEK WORD	Returns a two byte number (-32768 to 32767)
PEEK LONG*	Returns a four byte number (*32 bit versions)

EXAMPLE X=VARPTR(A\$) <---Get a safe place in memory to play with

```
:  
POKE X, 10  
POKE WORD X+1, 12000  
:  
PRINT PEEK(X)  
PRINT PEEK WORD(X+1)
```

RUN

```
10  
12000
```

REMARK See POKE, POKE WORD and POKE LONG, USR, MACHLG, CALL, LINE, HEX\$, OCT\$, UNS\$ and the section in the front of this manual; "Machine Language".



Important Note: This function is for people experienced with machine language and the hardware of their computer.



*Macintosh: Always use LongIntegers for expressions to pass an address or to retrieve a four byte LongInteger. See appendix.

MSDOS: An extra parameter is available to determine the segment of the variable: PEEK[WORD] (address,segment). Also see MEM and DEF SEG in the appendix.

statement PLOT

FORMAT **PLOT** *expr1,expr2* [**TO** *expr3,expr4...*]
 PLOT [TO] *expr1,expr2* [**TO** *expr3,expr4...*]

DEFINITION The PLOT statement is used to draw either one graphic point, or a line between two or more points, in the current COLOR. Examples:

```
PLOT 10,12                            <-- PLOT one point at position 10,12
PLOT 10,12 TO 100,100                <-- PLOT a line from 10,12 to 100,100
PLOT 10,12 TO 10,90 TO 1,1          <-- PLOT two lines: 10,12 to 10,90, to 1,1
PLOT TO 10,12                         <-- PLOT a line from last position to 10,12
```

EXAMPLE

```
CLS
MODE 5                                 <---Set graphics mode
PLOT 209, 304                         <--- Plots one pixel
:
COLOR -1                               <--- Sets COLOR to foreground
REM PLOT and angle
PLOT 209,304 TO 987, 643 TO 322,742
END
```

RUN

See illustrations on the following page.

REMARK As with all other ZBasic graphic commands, Device Independent Graphic coordinates of 1024 by 768 are the default. Expressions are truncated to an integer. Character type graphics will be substituted on computers, or modes, without graphic capabilities.

Also see CIRCLE,BOX,FILL,POINT,COLOR.



Macintosh: Use COORDINATE WINDOW to set the pixel graphics. Use COORDINATE to set your own relative coordinates or to set back to 1024x768. The upper left-hand corner of a WINDOW is coordinate 0,0.

MSDOS: Use COORDINATE WINDOW to set pixel coordinates. See COORDINATE to set relative coordinates or to set back to ZBasic coordinates.

Z80: POKE \$xx3F, &C9 for pixel coordinates. POKE \$523F, &C3 to set back to ZBasic coordinates. xx= CP/M=01, TRS-80 model 1,3=52, TRS-80 model 4=30.

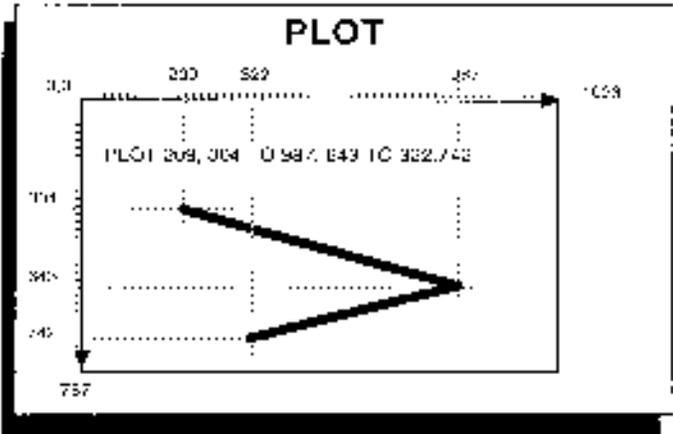
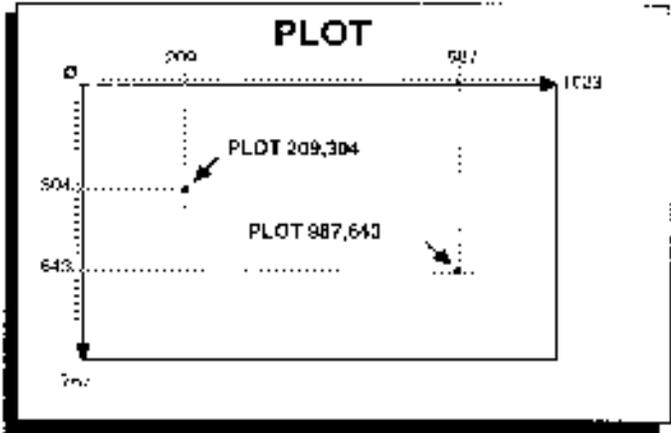
Apple // ProDOS: POKEWORD &85, 0 for pixel coordinates. Use MODE to set back to ZBasic coordinates.

Apple // DOS 3.3: POKE &F388,&60 for pixel coordinates. POKE &F388, &A9 to set back to ZBasic coordinates.

PLOT statement



PLOT continued

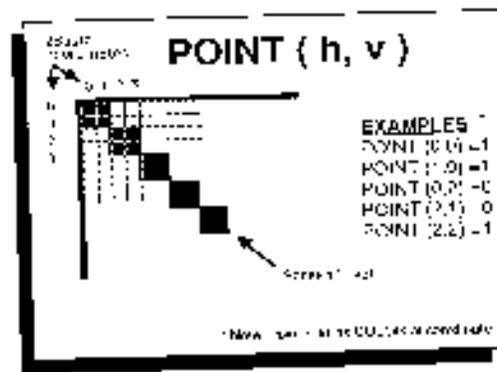


function POINT

FORMAT POINT (*expression*₁, *expression*₂)

DEFINITION Point is available on many computers to inquire about the COLOR of a specific screen graphic position. As with other commands, ZBasic Device Independent Graphic coordinates may overlap pixels.

In the example: 0=Background (white here), 1 =Foreground (black here)



As with all other ZBasic graphic commands, the device independent coordinate system of 1024 X 768 is the default.

EXAMPLE COLOR 1
PLOT 0,0 to 900,767
PRINT POINT(0,0)

RUN

1

REMARK If the coordinate is outside screen coordinates, a -1 will be returned.

See COLOR,BOX,CIRCLE and the section; "Graphics".

See COORDINATE or PLOT for ways of converting some versions of ZBasic to pixel coordinates that can used with POINT.



POINT is not available for CP/M versions (including Kaypro graphic versions).

POKE statement

FORMAT **POKE [WORD]** *expression%,expression2*
POKE LONG *expression&,expression2&**

DEFINITION POKE writes the value of expression2 into a memory location. The first expression is the address to POKE. The expression2 is the data to POKE.

<u>TYPE</u>	<u>expression2</u>
POKE	One byte
POKE WORD	Two bytes
POKE LONG*	Four bytes (*32 bit machines only)

EXAMPLE

```
X = 12345: XA = VARPTR(X)
PRINT"Byte at ";UNS$(XA);" =";PEEK(XA)
:
POKE XA,99
PRINT"Byte at ";UNS$(XA);" =";PEEK(XA)
:
POKE WORD XA,44444
PRINT"WORD at ";UNS$(XA);" =";UNS$(PEEK WORD(XA))
END
```

RUN

```
Byte at 59009 = 57
Byte at 59009 = 99
Byte at 59009 = 44444
```

REMARK Also see PEEK,PEEK WORD,PEEK LONG,MACHLG,CALL,LINE and the chapter "Machine Language" at the beginning of this manual.



Important Note: Indiscriminate use of this command may cause unpredictable computer operation and loss of data or program. This statement is for experienced machine language programmers only. Porting of programs with POKE is not recommended.



***Macintosh:** Always use LongIntegers for addresses and when using POKE LONG or PEEK LONG.

MSDOS: There is an optional parameter for segment:
POKE[WORD] *address, data, segment*. See MEM and DEF SEG in the appendix.

FORMAT **POS** (*byte expression*)

DEFINITION Returns the current horizontal cursor position, from zero to 255, for a screen printer or disk file.

The expression specifies a device as follows:

- POS(0) Default device (normally the video monitor)
- POS(1) Printer
- POS(2) Disk file (limited to one file using carriage returns)

EXAMPLE

```
CLS
PRINT "READ and DISPLAY SCREEN POS"
FOR I = 0 TO 30 STEP 10
  PRINT TAB(I); POS(0)
NEXT
:
PRINT "READ and DISPLAY PRINTER POS"
DEFTAB 5
FOR I = 0 TO 6
  LPRINT,
  PRINT POS(1)M
NEXT
END
```

RUN

```
READ and DISPLAY SCREEN POS
0          10          20          30

READ and DISPLAY PRINTER POS
6    12    18    24    30    36
```

REMARK A carriage return will set the POS value to zero. PAGE will return the current line position for the printer.

Also see WIDTH,PAGE and WIDTH LPRINT.

While this command will work the same on all systems, it is dependent on screen and printer widths.

PRINT# statement

FORMAT **PRINT #** *expression, list of things to print.....*

DEFINITION Used to PRINT information to a disk file or other device in text format. Numbers or strings will appear in the file or device similar to how they would look on the screen or printer.

The expression is the file number assigned to a disk file or other device in an OPEN statement.

INPUT# or LINEINPUT# are normally used to read back data created with PRINT# (although READ# may also be used).

EXAMPLE

```
A$="TEST":B$="TEST2":C=900
:
OPEN "O",1,"TEST.DAT"
PRINT#1,"HELLO","A$","B$","C <--- Quoted comma delimiters for INPUT#
CLOSE#1
:
OPEN "I",1,"TEST.DAT"
INPUT#1,X$,Y$,Z$,A%      <--- INPUT in same order and same type
:
PRINT X$,Y$,Z$,A%
:
CLOSE#1
END
```

RUN

```
HELLO            TEST            TEST2            900
```

REMARK While this command will work the same on all systems, it is dependent on disk input/output capabilities. Use INPUT# or LINEINPUT# to read back data written with PRINT#.

Be sure to see the entry on INPUT# in this reference section for more information about using PRINT# and INPUT# together and also information about MSBASIC syntax differences.

See ROUTE, OPEN, OPEN"C", INPUT#, LINEINPUT#, READ#, WRITE#, LPRINT and the section in the front of this manual called "Files" for more information.

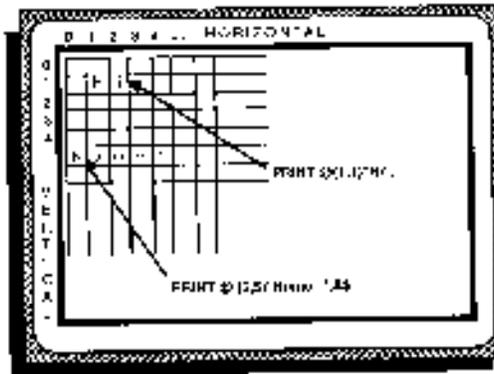
statement PRINT

FORMAT **PRINT** [{@|%} (expr1, expr2)] [list of things to print....]

DEFINITION The PRINT statement is used to output information to the current device, normally the video.

@ (expr1,expr2) Specifies text coordinates.
% (expr1,expr2) Specifies graphic coordinates.
Note: Expr1=Horizontal. Expr2=Vertical.

EXAMPLE



```
PRINT@(1,1) "Hi" ;  
PRINT@(0,5) "Name:" ;A$  
END
```

REMARK PRINT followed with a semi-colon will disable the carriage return.

A PRINT item followed by a comma will cause the next element to be printed at the next tab stop defined by DEF TAB.

While this command will work the same on all systems, it is dependent on hardware.

See ROUTE for ways of sending PRINT data to another device like a printer, disk file or serial port.

See "Screen and Printer Text Control" in the front section of this manual for other ways of formatting text.

As with all other ZBasic graphics commands, PRINT %(x,y) defaults to printing at the position specified by the Device Independent Graphic coordinates of 1024 x 767. See PLOT or COORDINATE for ways of changing some versions of ZBasic to using other coordinates.

PRINT USING function

FORMAT PRINT[# *filename*,] **USING** *formatstring* ;*numeric expression*:[**USING**...]

DEFINITION This function permits formatting numeric data in PRINT or PRINT# statements.

The last numeric digit displayed will be rounded up by adding 5 to the first digit on the right that is not displayed.

The *formatstring* may be a quoted or string variable using the following symbols:

<u>Symbol</u>	<u>Definition</u>
#	Holds place for a digit. More than one may be used. An example of using this symbol to hold dollars and cents: PRINT USING "\$###.##" ;A# \$123.45
,	Insert a comma in that place. An example of using it to format numbers with dollars and cents would be: PRINT USING"\$##,###.##" ;A# \$12,235.67
.	Determines placement of decimal point within the format field: PRINT USING"\$##,###,###.##" ;A# \$12,345,678.90
\$	Prints a dollar sign on the left of the format. See examples above.
+	Prints a floating plus or minus sign on the side of the number where the plus sign holds the place. PRINT USING"+####.##" ;A# +1234.56 PRINT USING"+####.##" ;-1234.56 -1234.56
-	Prints a minus sign only if the expression is negative. PRINT USING"+####.##" ;A# 1234.56 PRINT USING"+####.##" ;-1234.56 -1234.56
*	Fill the spaces before a number with asterisks. One example would be formatting output when printing checks. PRINT USING"\$##,###,##.##" ;12.34 \$*****78.90

EXAMPLE See examples on next page...

REMARK When **error** is printed in the format field, this indicates the occurrence of an overflow condition and replaces the number that would have been printed. An overflow condition is when the value of the expression used would have exceeded the boundaries of the format.

USING not available for string formatting. See LEFT\$,RIGHT\$,STRING\$ and MID\$.



This version allows USING without PRINT. A\$=USING"#####.##";232 is acceptable. See appendix for additions to exponential formatting with this version.

PRINT USING continued

FORMAT EXAMPLES

In all the examples A=12345.678. Note that .678 rounds up to .68.

<u>PRINT USING FORMAT</u>	<u>RESULT</u>
"*\$###,###,###,###.##";A	*****\$12,345.68
"%###.##";A/1000	%12.3
"+###,###.##";A	+12,345.68
"-###,###.##";-A	-12,345.68
"##/##/##";A	1/23/45
"##:##:##";A	1:23:45
".###,###,###,###";1.345E-8	.000,000,013,450
".#####";1.345E-8	.00000013450
"###,###,###,###,###";9.123E15	9,123,000,000,000,000
"###.##E16";123E15*1E-16	12.30E16

PROGRAM EXAMPLE

```
A$="##.##"
:
PRINT USING A$;10.2,USING A$;9.237, USING A$; 4.555
PRINT 10,12,13, USING A$;12.399
:
PRINT@(0,10);USING A$;23.12321
:
PRINT%(0,295);USING "@#####.##";12.33
:
OPEN"O",1,"TESTFILE"
PRINT#1, USING A$;9.999
CLOSE#1
```

RUN

```
10.20          9.24          4.56
10             12             13             12.40

23.12          <--- at text position 0.10
@12.33 <--- at graphic position 0,295
10.00          <--- To disk file "TESTFILE"
```

PSTR\$ function/statements

FORMATS function
 PSTR\$(var%)

 statements
 READ PSTR\$(var%)
 PSTR\$(var%) = "quoted string constant"

DEFINITION The statements load the address of a string constant into var%.

The function returns the string pointed to by var%.

EXAMPLE DATA Andy, Dave, Scott, Mike
 :
 DIM D(4)
 :
 FOR X=1 TO 4 <---Set Pointer String to DATA items above
 READ PSTR\$(D(X))
 NEXT
 :
 "Print PSTR\$ of D(n)"
 FOR X=1TO4
 PRINT PSTR\$(D(X))
 NEXT
 END
 :
 PSTR\$(g%)="Hello" <--- Set Pointer String to a constant
 PRINT PSTR\$(g%)

RUN

Andy
Dave
Scott
Mike
Hello

REMARK This is a handy way to save string memory. Examples:

A\$="Hi There!"
A\$ will take at least 10 bytes (256 bytes if not defined). The quoted string takes another 10 bytes. Total memory used: 20 bytes

PSTR\$(A)="Hi There!"
The quoted string "Hi There!" takes 10 bytes. The integer variable "A" takes two bytes. Total memory used: 12 bytes



Macintosh: Use var& instead of var%.

statement PUT

FORMAT PUT(x1,y1) variable [(array index[, array index[,...]] [,mode]

DEFINITION This statement places the graphic bit image stored in a array with the GET statement, to the screen position at coordinates specified by x1,y1.

If an array has been used then you MUST specify the index number of the array (some versions of BASIC always assume an integer array. ZBasic will allow you to store bit images in any variable type as long as enough memory is available to do so.

Memory required for pixel images is calculated using this formula (based on GET(x1,y1)-(x2,y2) where x1 and y1 designate the upper right-hand-corner of the image and x2 and y2 are the pixel positions designating the lower-left-hand-corner of the image):

$$6+((y2-y1)+1) * ((x2-x1+1) * bpp+7)/8)$$

The number of bits per pixel (bpp) depends on system colors or grey levels. See next page for specifics. Also see GET in this reference section, for detailed information about storing the pixel image in an array.

mode	XOR	XORs the pixels over the background pixels. This is the most useful for animation purposes and is also the default.
	OR	ORs the pixels over the existing pixels. This one way to cover the background graphics (overlays the existing graphics).
	AND	ANDs the picture with background.
	PRESET	Similar to PSET except the reverse image is shown (negative).
	PSET	Draws the image over the background exactly as created.

It is recommended that COORDINATE WINDOW be used when using GET.

EXAMPLE

```
DIM A(10000)
MODE 7 <---- Not needed on the Macintosh version
COORDINATE WINDOW <---- Pixel coordinates
:
CIRCLE 100,100,80
GET (0,0)-(100,100), A(1)
:
FOR x= 1 TO 200 STEP 3
  PUT (x, 90), A(1) <----Do it twice to XOR the pixels and move the image across
  PUT (x, 90), A(1) the screen without disturbing the background
NEXT x
:
END
```

This routine moves a section of a circle across the screen. It is XORed to the screen twice so the item doesn't repeat and it will appear to move across the screen without disturbing the background (default PUT mode is XOR).

continued...

PUT statement

REMARKS It is important to see entry under GET for more information.



Macintosh: With this version of ZBasic, PUT has another, optional, parameter: PUT (x1,y1) [-(x2,y2)], var. The second parameter allows you to scale the image, making it either larger or smaller by giving the rectangle size in which it is to appear. The x2, y2 parameter is the lower-right corner of the image.

Bits-per-pixel (bpp) will vary by the type of Macintosh you have. The standard black and white Macintoshes have one bit-per-pixel.

The Macintosh II may have up to 16 bits-per-pixel (with up to 256 colors or grey-levels per pixel). Check addendum of Macintosh II for specifics.



MSDOS: Bits-per-pixel (bpp) will vary with the graphics adaptor board being used:

<u>GRAPHIC TYPE</u>	<u>MODE(s)</u>	<u>COLORS</u>	<u>BITS PER PIXEL (bpp)</u>
CGA	5	4	2
CGA	7	2	1
EGA	16-19	3-16	2 (64K or less on EGA card)
EGA	16-19	16	4 (More than 64K on card)
HERCULES	20	1	1



Z80: GET and PUT are not supported with these versions of ZBasic.



Apple // ProDOS and DOS 3.3: GET and PUT are not supported with this version. See DRAW example on ProDOS disk and the BLOAD and BSAVE functions for possible alternatives.

command QUIT

FORMAT **QUIT**

DEFINITION QUIT is used to exit the ZBasic Standard Line editor and return control to the operating system.

EXAMPLE **QUIT**

DOS Ready <----DOS prompt of your System.

REMARK We highly recommend saving your program prior to using **QUIT**.



Macintosh: You may also quit from the menu.

MSDOS: SYSTEM functions the same as QUIT.

RANDOM statement

FORMAT **RANDOM** [**IZE**] [*expression*]

DEFINITION Seeds the random number generator so that ZBasic produces a new sequence of random numbers.

If *expression* is used, the RND function will return a repeatable series of numbers.

EXAMPLE

```
DEFTAB 5
RANDOM 12345
FOR I = 1 TO 5
  PRINT RND(10),
NEXT I
:
RANDOM 12345 <--- Let's see if it repeats as above.
FOR I = 1 TO 5
  PRINT RND(10),
NEXT I: PRINT
```

RUN

```
8      1      10      4      7
8      1      10      4      7
```

```
PRINT"Press any key to set random seed" <--- Paranoid seed routine
DO
  R=R+1
UNTIL LEN(INKEY$)
RANDOM R
:
FOR I = 1 TO 5
  PRINT RND(10),
NEXT I
END
```

RUN

```
Press any key to set random seed
1      8      8      5      9
```

REMARK

The results of the first two passes were the same because the seed of 12345 was the same. When a different number is used, or no number, the result will be RANDOM.

If *expression* is the same, the same random pattern will be repeated with all versions of ZBasic.



The [IZE] part of RANDOM is not supported on the Apple // and Z80 versions.

statement RATIO

FORMAT RATIO *byte expression1, byte expression2*

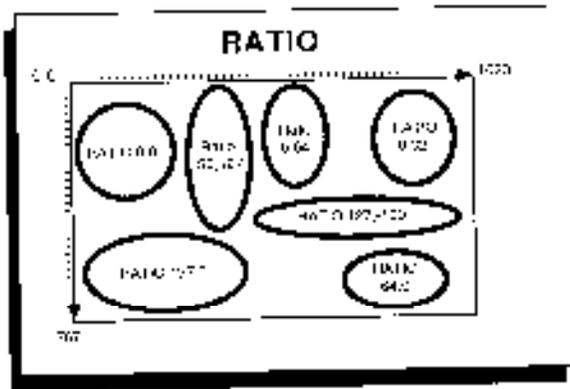
DEFINITION This statement will change the aspect ratio of graphics created with CIRCLE.

byte expression1 Horizontal ratio. A number between -128 and +127 that gives the relationship of the width of the circle to normal (zero).

byte expression2 Vertical ratio. A number between -128 and +127 that gives the relationship of the height of the circle to normal (zero).

<u>Value</u>	<u>Relationship to normal</u>
+127	= 2.0 times normal
+64	= 1.5 times normal
+32	= 1.25 times normal
0	= 0 Normal proportion
-32	= 0.75 times normal
-64	= 0.5 times normal
-96	= 0.25 times normal
-128	= 0 times normal (no width or height)

EXAMPLE



RATIO -50, 127
CIRCLE h,v,r

REMARK RATIO settings are executed immediately and all CIRCLE commands, including CIRCLE TO and CIRCLE PLOT will be adjusted to the last RATIO.



Also see ROUNDRECT toolbox routines for other options to creating circles with various rations.

READ# statement

FORMAT **READ #** *filename*, {*var* |*var*;*stringlength* } [, ...]

DEFINITION Reads strings or numbers saved in compressed format with WRITE# and stores them into corresponding variables. The list may consist of any type string or numeric variables or array variables.

<i>filename</i>	The filename to work from
<i>var</i>	Any numeric type variable
<i>var</i> ;	String variable
<i>stringlength</i>	The number of characters to load into the string variable



Important Note: A string variable must be followed by ;*stringlength* to specify the number of characters to be read into that string.

EXAMPLE

```
REM The four variables below will require 18 bytes for storage
REM A$=4 bytes, A!= 4 bytes, A#=8 bytes, A%=2 bytes
:
A$="TEST": A!="12345.6":A#="12345.67898":A%=20000
:
OPEN"0",1, "DATAFILE", 18      <--- Write a file with a record length of 18
WRITE #1, A$;4, A!, A#, A%
CLOSE#1
:
OPEN"I" ,1,"DATAFILE", 18
READ#1, Z$;4, Z!, Z#, Z%      <--- Read in same order and type (see notes)
CLOSE# 1
:
PRINT Z$, Z!, Z#, Z%
END
```

RUN

```
TEST                    12345.612345.67898                    20000
```

REMARK Note: Do not mix variable types when using READ# and WRITE#. Reading string data into numeric variables, and visa-versa, will create variables with incoherent data.

READ# and WRITE# store and retrieve numeric data in a compressed format. This saves disk space and speeds program execution.

While you may load numeric data into strings and convert using CVB or CVI, it is best to refrain from this since it requires more time and is less efficient.

See the chapter "Files" for more detailed information using random and sequential files. Also see RECORD, LOC,REC,LOF and "Disk Error Trapping".

statement READ

FORMAT READ [*variable* {-or- PSTR\$(*var%*)}[,...]]

DEFINITION The READ statement reads strings or numbers from a DATA statement into corresponding variables.

The variable list can consist of any combination of variable types (string or numeric, including arrays).

If no variable is given the READ statement will skip one DATA item.

EXAMPLE

```
DIM P%(3)
:
DATA Joe, Smith, Harry, "@ Cost"
DATA 1234.5, 567.8, 91011.12, 1314.15
:
READ A$, B$, C$, D$ <--- Regular old fashioned READ
READ A!, B!, C!, D!
PRINT A$, B$, C$, D$
PRINT A!, B!, C!, D!
:
RESTORE <--- Set pointer back to start of DATA to READ again
FOR X=0 TO 3
  READ PSTR$(P%(X)) <---Use pointer string to point at DATA string constants
NEXT:PRINT
PRINT "PSTR$>"
FOR X= 0 TO 3
  PRINT PSTR$(P%(X)),
NEXT
:
RESTORE 6 <--- Set DATA pointer to the sixth item
READ A#
PRINT A#
END
```

RUN

```
Joe           Smith           Harry           @ Cost
1234.5 567.8   91011.12       1314.15

PSTR$> Joe           Smith           Harry           @ Cost
567.8
```

REMARK



Leading spaces in string data statements will be ignored unless contained in quotes.

Do not read numeric data into string variables and vice versa (no error is generated).
Don't read past the end of a data list.

See RESTORE,PSTR\$ and DATA.

RECORD statement

FORMAT **RECORD** [#] *filename*, *recordnumber* [, *location in record*]

DEFINITION The RECORD statement is used to position the file pointer anywhere in a file. Once the file pointer has been positioned you may read or write data from that position.

RECORD can position both the RECORD pointer and the location within a record.

filename Filename from 1 to 99

recordnumber RECORD number to point to. Default is zero.

location in record Optional location in RECORD. Default is zero.

EXAMPLE `OPEN"R",1,"TESTFILE",30`
 `:`
 `FOR Position = 0 to 29`
 `RECORD #1, 6, Position`
 `READ#1, A$;1` <--- Reads one character at a time from record 6.
 `PRINT A$;`
 `NEXT`
 `:`
 `CLOSE#1`
 `END`

 RUN

 Fred Stein

See illustration next page...

REMARK The default RECORD length is 256 bytes. The maximum record length is 65,535. The maximum number of records in a file is 65,535.

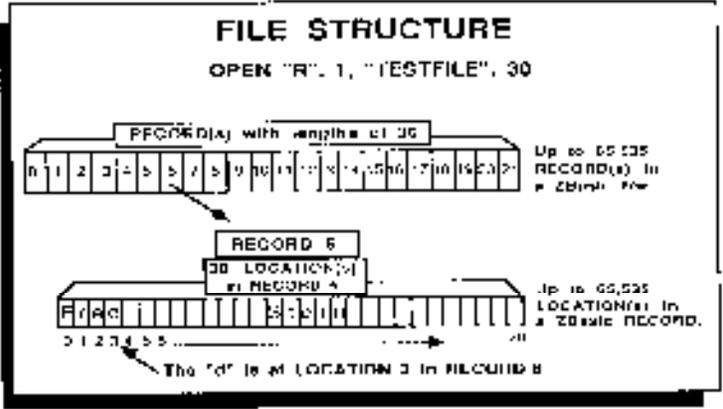
See OPEN,READ#,WRITE#,PRINT#,INPUT#,LINEINPUT#,LOC,LOF, REC, CLOSE, and the chapter entitled "Files".



The maximum record length and number of records in a file is 2,147,483,647.

statement RECORD

RECORD continued



In the illustration, the name "Fred Stein" was stored in RECORD six of "TESTFILE".

To point to FILE #1, RECORD 6, LOCATION 3 use the syntax:

```
RECORD# 1, 6, 3
```

The location within a record is optional (zero is assumed if no location is given).

If RECORD 1, 6 had been used (without the 3), the pointer would have been positioned at the "F" in "Fred".

If RECORD is not used, reading or writing starts from the current pointer position. If a file has just been opened, the pointer is positioned at the beginning.

After each read or write, the file pointer is moved to the next position in the file.



The maximum record length and number of records in a file for this versions is 2,147,483,647.

REC function

FORMAT **REC** (*filename*)

DEFINITION Returns the current position of the record pointer for the file specified by expression. The first record in a file is record zero (0).

Also often used with REC is LOC which returns the position within the record.

EXAMPLE `OPEN "O",1,"THISPROG",10 <--- Record length of ten`
 :
 `A$="012345" <--- String length of six`
 :
 `FOR I = 0 TO 3`
 `PRINT#1, A$;`
 `PRINT "On pass";I;" file position was ";`
 `PRINT "Rec="REC(1);" and LOC=";LOC(1)`
 :
 `CLOSE#1`
 `END`

RUN

On Pass 0 file position was REC=0 and Loc=6
On Pass 1 file position was REC=1 and Loc=2
On Pass 2 file position was REC=1 and Loc=8
On Pass 3 file position was REC=2 and Loc=4

Right after the middle RECORD statement; REC=0 and LOC=4

REMARK The default record length is 256 bytes. LOC returns the position within a RECORD.

See OPEN,CLOSE,LOC,LOF,RECORD,READ#,WRITE# and the chapter entitled "Files".

statement REM

FORMAT **REM** followed by programming remarks

DEFINITION The REM statement is used for inserting comments or remarks into a program. ZBasic ignores everything following a REM statement.

To save time, you can type an apostrophe (') at the beginning of a line and it will be converted into a REM statement.

EXAMPLE

```
REM            This is a comment or remark
REM            ZBasic ignores everything following a REM
REM            Including any commands embedded in the remark
:
REM            Colons are often used to make blank lines.
:
:
:
REM            Thoughtful use of REM makes a program easier to read.

RUN

ZBasic Ready_
```

REMARK REM statements are not compiled and do not take up any memory in the object code.

Note: Some versions of ZBasic will not convert the apostrophe to REM.

RENAME statement

FORMAT **RENAME** *string1* {[,|TO]} *string2*

DEFINITION This statement is used to rename the file *string1* to the new name *string2*.

EXAMPLE **DIR**

```
GOOGOO            ZBASIC.COM
FRED.BAS            OLDFILE.BAS
```

```
INPUT "FILE NAME TO CHANGE: ";File1$
INPUT "NEW NAME FOR FILE: ";File2$
RENAME File1$ TO File2$
```

RUN

```
FILE NAME TO CHANGE: GOOGOO
NEW NAME FOR FILE: GOONIE
```

DIR

```
GOONIE            ZBASIC.COM
FRED.BAS            OLDFILE.BAS
```

REMARK This command is also available in command mode. Remember that filename formats are different from system to system and may not be available for some machines.



TRS-80 model 1,3: RENAME not supported with these versions.



Macintosh: Pathnames or volume number may be used.

Macintosh: RENAME file1\$ {TO|,} file2\$ [, *volume number*%]. Also see NAME.

MSDOS: See CHDIR, PATH\$, RMDIR and MKDIR in the MSDOS appendix for controlling pathnames and directories. Also see NAME.

Apple // ProDOS: Pathnames supported.

command RENUM

FORMAT RENUM [*new*][,*old*][, *increment*]

DEFINITION Used for renumbering program lines.

new The first new assigned line number desired after renumbering is complete. default = 10
old The first old line where you want renumbering to begin. default = 0
increment The increment between line numbers. default = 10 (256 maximum)

If an argument is omitted the default will be used.

This command will automatically update line references (GOTO,GOSUB, etc). If a line reference is to a non-existent line, it will use the next existing line number.

EXAMPLE

```
7 IF I = 200 THEN 567
74 PRINT I
197 I = I + 1: GOTO 74
567 END
```

RENUM
LIST

```
10 IF I = 200 THAN 40
20 PRINT I
30 I = I + 1: GOTO 10
40 END
```

REMARK Line increments are limited to 256. If you issue a RENUM command that exceeds the number of allowable lines (65,534) , an error will occur and your text will be unaltered.

If you are unsure of what the results may be, SAVE your program BEFORE renumbering!



Some versions offer options for using, or not using, line numbers with full screen editors. Check your appendix for specifics.



See RENUM*,UNNUM,INDENT and FIX in the MSDOS appendix for other options.

RESET statement

FORMAT RESET

DEFINITION Closes all open files and devices. Functionally identical to CLOSE without parameters.

EXAMPLE

```
OPEN"O",1,"FRED"  
OPEN"I",2,"HARRY"  
  
IF ERROR THEN RESET  
  
END
```

REMARK See CLOSE



Not supported on Apple // or Z80 versions of ZBasic. Simply use CLOSE without a filename to close all open files.

statement RESTORE

FORMAT **RESTORE** [*expression*]

DEFINITION This statement resets the DATA pointer to the first DATA statement or optionally to the DATA item specified by expression.

If the expression is omitted, the first DATA item is assumed. ZBasic automatically sets the pointer to the next item after each variable is READ.

EXAMPLE

```
DATA ZERO, ONE, TWO, THREE, FOUR, FIVE
DATA SIX, SEVEN, EIGHT, NINE, TEN
:
"Start"
DO
  INPUT "What item do you want" "Item%"
  IF (item%<0) OR (item%>10) THEN "Start"
  RESTORE Item%
  READ A$
  PRINT "Item number: ;Item%;" is: ";A$"
UNTIL Item%=0
:
RESTORE                             <--- Set to beginning of DATA
READ A$: PRINT A$
:
END
```

RUN

```
What item do you want: 4
Item number 4 is: FOUR
```

```
What item do you want: 9
Item number 4 is: NINE
```

```
What item do you want: 0
Item number 0 is: ZERO
```

```
ZERO
```

REMARK If an attempt is made to READ or RESTORE past the last DATA item, the result will be zeros or NULL strings. No error will be returned.

Also see READ, PSTR\$ and DATA.

RETURN statement

FORMAT **RETURN** [*line*]

DEFINITION The RETURN statement is used to continue execution at the statement immediately following the last executed GOSUB or ON GOSUB statement.

If optional line is used, the last GOSUB is POPPED off the stack and a GOTO line is performed.

EXAMPLES GOSUB "First"
 :
 "Second"
 PRINT "RETURN comes here."
 END
 :
 "First"
 PRINT "This is a subroutine"
 RETURN

RUN

This is a subroutine
Return comes here

```
GOSUB "Routine"  
END  
:  
"Weird"  
PRINT"Ended Here!"  
STOP  
:  
"Routine"  
PRINT"At 'Routine'"  
RETURN "Weird"
```

RUN

At 'Routine'
Ended Here!

REMARK When ZBasic encounters a RETURN statement which was not called by a GOSUB, it will return to the program that executed it (either DOS or the ZBasic editor).

Using RETURN line WITHOUT A GOSUB or from the middle of a LONG FN will cause unpredictable (probably disastrous) system errors.



Use caution when using RETURN line to exit event trapping routines like DIALOG ON,MENU ON, TRON,BREAK ON...

function RIGHT\$

FORMAT **RIGHT\$(string, expression)**

DEFINITION Returns the right-most expression characters of string.

EXAMPLE

```
A$="HELLO"  
:  
FOR I = 0 TO 6  
  PRINT I, RIGHT$(A$,I)  
NEXT I  
:  
A$ = "JOHN DOE"  
:  
SP = INSTR(1,A$," ")  
PRINT"LAST NAME:",  
PRINT RIGHT$(A$,LEN(A$)-SP)  
:  
END
```

RUN

```
0  
1            0  
2            LO  
3            LLO  
4            ELLO  
5            HELLO  
6            HELLO  
LAST NAME :   DOE
```

REMARK If expression is more than the characters available, all the characters will be returned.

See LEFT\$,VAL,STR\$,STRING\$,SPACE\$,SPC, MID\$ and the chapter entitled "String Variables" in the front section of this manual.

RND function

FORMAT **RND** (*expression*)

DEFINITION The RND function returns a random integer number from 1 to expression.

EXAMPLE

```
RANDOM
A=9
:
FOR I=1 TO 5
  PRINT RND(A),
  PRINT RND(10000)*.0001
NEXT I
:
END
```

RUN

```
3           .9201
7           .8211
1           .0912
2           .7821
9           .0108
```

REMARK Some versions of BASIC return a floating point random number between 0 and 1; use RND(10000)*.0001 to emulate this (it will slow down execution).

Also see MAYBE and RANDOM.

If the same speed number is used for RANDOM, the random numbers generated by RND will be predictable on the all versions of ZBasic.

The largest number you may use for a RND *expression* is 32,767.

statement ROUTE

FORMAT **ROUTE** [#] *expression*

DEFINITION This statement is used to route PRINT statements to a specified device. The following are the values to be used as expression.

<u>Device number</u>	<u>Routes PRINT statements to</u>
negative numbers	I/O devices; See your appendix for specifics.
0	Screen (default)
1-99	Disk files specified by number
128	Printer

EXAMPLE

```
ROUTE 128
PRINT "HELLO"           <--- This HELLO goes to the printer
:
OPEN"O",1,"Test"
ROUTE 1
PRINT "HELLO"           <--- This HELLO goes to file "Test"
CLOSE#1
:
OPEN"C",-1,300
ROUTE -1
PRINT "HELLO"           <--- This HELLO goes to a serial device
CLOSE#-1
:
ROUTE 0
PRINT"HELLO"           <--- This HELLO goes to the screen
END
```

RUN

HELLO

REMARK You should eventually route the output back to a screen device (ROUTE 0).

See PRINT,OPEN"C" and the chapter "Files" for more information.



Also see ROUTE 128, CLEAR LPRINT, DEF LPRINT and DEF PAGE for more information about routing text and graphic output to the Imagewriter and Laserwriter. Be sure to use CLEAR LPRINT with ROUTE 128 to tell the Macintosh printer driver to print the page.

RUN statement

FORMAT **RUN** [*filename*]

DEFINITION The RUN statement does one of two things.

RUN *filename* Loads a compiled chain program specified by filename and executes it:

```
OPEN "I", 1, "Prog.CHN"  
RUN 1
```

RUN Clears all variables and pointers and restarts the current program from the first line.

EXAMPLE OPEN "I", 2, "MENU"
RUN 2

<---Loads and RUNS CHAIN program "MENU"

```
TRONB  
FOR X=1 TO 100  
  PRINT X  
NEXT  
RUN
```

<--- RUNS this program over and over...

REMARK Also see the RUN command and the chapters "Running ZBasic Programs" and "Chaining" for more information.



Also see RUN filename\$, volumenumber% in the appendix.

FORMAT **RUN** [{"+}*"}] *filename* [""]

DEFINITION This command is used from the Standard Line Editor to compile a program:

RUN	Compiles source code in memory and executes.
RUN <i>filename</i>	Compiles source code called <i>filename</i> from disk and executes. Source code must have been saved in tokenized format with SAVE (not as a text file).
RUN*	Compiles source code in memory and saves as a stand-alone application on disk. Asks for filename after compiling.
RUN* <i>filename</i>	Compiles source code called <i>filename</i> from disk and saves as a stand-alone application on disk. Source code must have been saved tokenized (not as a text file). Asks for filename after compiling.
RUN+	Compiles source code in memory and saves as a chain file to disk (no runtime included). Asks for filename after compiling.
RUN+<i>filename</i>	Compiles source code called <i>filename</i> from disk and saves as a chain file to disk (no runtime included). Asks for filename after compiling.

EXAMPLE `PRINT "THE PROGRAM RUNS!"`

RUN

`THE PROGRAM RUNS!`

REMARK Compiling from disk will destroy any text currently in memory. If an error is encountered when compiling from disk, ZBasic will load the source code and print an error message.

After a successful compilation, typing MEM will return memory used for the object code and variables.

See "Executing Programs" in the front of this manual for more information about compiling large programs.



Also see COMPILE and LCOMPILE for ways of compiling a program and seeing all the compile time errors at once (instead of one at a time as with RUN).

SAVE command

FORMAT **SAVE** [{"*|+"} ["] *filename* [""]]

DEFINITION **SAVE** is used from the Standard Line Editor to save the source code in memory. You may save your source code in a number of formats:

SAVE Saves program in tokenized format. This requires less room on the disk and saving and loading is much faster than with text files. In order to compile a file from disk a program must be saved in this format.

SAVE* Saves program in TEXT or ASCII format. This allows you to load the program into other word processors or editors. Loads more slowly than SAVE above.

SAVE+ Same as SAVE* but line numbers are removed. Be sure your program doesn't use label references with GOTO, GOSUB or other commands, since when a program is re-loaded, line numbers are added back in increments of one which will make line number references incorrect.

Note: Source code is the program you type in. Object code is the machine language program created when you compile the source code with RUN. See RUN for more information about compiling and saving compiled programs to disk.

EXAMPLE `SAVE* PROGRAM.TXT` <---SAVE program in ASCII (text)
`SAVE AR.BAS` <---SAVE program tokenized (condensed)
`SAVE+ FILE.TXT` <---SAVE program in ASCII - with no line numbers

REMARK Also see LOAD,APPEND,MERGE and RUN.

statement SELECT

FORMAT **SELECT** [*expression or simplestring*]
 CASE [IS] relational condition [, relational condition][,...]
 statements...
 CASE [IS] condition [, condition][,...]
 statements...
 CASE [IS] boolean expression
 statements...
 CASE ELSE
 END SELECT

DEFINITION Provides a structured and efficient way of doing multiple comparisons with a single expression. While IF-THEN or LONG-IF statements could be used, they are harder to follow when reading program listings.

EXAMPLE X=CARDTYPE:REM MSDOS Cardtype example.
 SELECT X
 CASE 0
 PRINT"CGA CARD":MODE 7
 CASE 1
 PRINT"EGA CARD":MODE 19
 CASE 2
 PRINT"EGA with Mono":MODE 18
 CASE 3
 PRINT"HERCULES CARD":MODE 20
 CASE 255
 PRINT "Monochrome Monitor":MODE 2
 CASE ELSE
 PRINT"No Video card installed"
 END SELECT

REMARK See CASE and END SELECT for more examples.



Important Note: Exit a SELECT structure only at the END SELECT.



SELECT is not supported with the Apple or Z80 versions of ZBasic. Use IF-THEN or LONG-IF to accomplish the same thing.

SGN function

FORMAT **SGN**(*expression*)

DEFINITION Returns the sign of *expression*.

If *expression* is:

Positive	+1 is returned.
Zero	0 is returned.
Negative	-1 is returned.

EXAMPLE

```
DEFDBL A-Z: DEFTAB 8: WIDTH 40
PRINT "      X", "ABS(X)", "INT(X)", "FRAC(X)", SGN(X) "
:
FOR X = -15.0 TO +15.0 STEP 3.75
  PRINT USING "-#.##";X,
  PRINT USING "##.##";ABS(X),
  PRINT USING "-#.##";INT(X),
  PRINT USING "-#.##";FRAC(X),
  PRINT USING "-#.##";SGN(X)
NEXT X
```

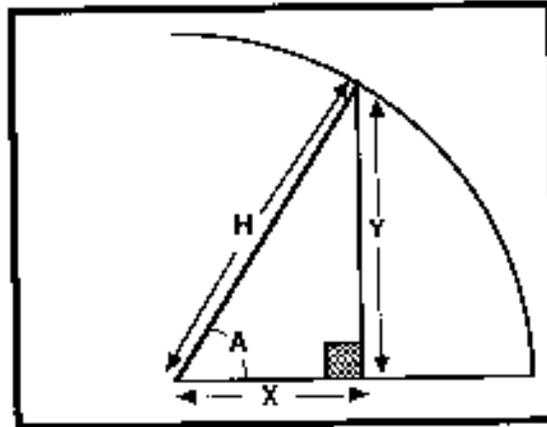
RUN

X	ABS(X)	INT(X)	FRAC(X)	SGN(X)
-15.00 15.00		-15.00	.00	-1.00
-11.25 11.25		-11.00	-.25	-1.00
- 7.50 7.50		- 7.00	-.50	-1.00
- 3.75 3.75		- 3.00	-.75	-1.00
.00 .00		.00	.00	.00
3.75 3.75		3.00	.75	1.00
7.50 7.50		7.00	.50	1.00
11.25 11.25		11.00	.25	1.00
15.00 15.00		15.00	.00	1.00

REMARK Also see UNS\$, FRAC, INT, ABS and negation.

FORMAT **SIN** (*expression*)

DEFINITION The SIN function returns the sine of the expression in radians.



$$\sin(A) = Y/H, \quad H \sin(A) = Y, \quad Y/\sin(A) = H$$

$$\sin(A) = Y/H, \quad H \sin(A) = Y, \quad Y/\sin(A) = H$$

EXAMPLE `X#=SIN(123)`
 `PRINT SIN(X2#)`

REMARK SIN is a scientific function. The precision for scientific functions may be configured. See "Configure" in the front of this manual for more information.

See the "Math" and "Expressions" sections of this manual and ATN, TAN, COS, EXP, SQR, ^.

INTEGER SINE: ZBasic provides a predefined USR function to do hi-speed integer sines. This speeds up sine speed by up to 30 times:

USR8(*angle*) returns the integer sine of *angle* in the range +-255 (corresponding to +-1). The angle must be in brads: See CIRCLE for examples of brads. Example:

```
MODE 7 :CLS
FOR I=0 TO 255
  PLOT I<<2, -USR8(I)+384
NEXT I
```

SOUND statement

FORMAT **SOUND** *frequency, duration*

DEFINITION **SOUND** may be used to create sound effects or music.

frequency Frequency 120 Hz to 10,000 Hz.

duration Duration in 1 millisecond increments.

Note: Hz (Hertz) represents cycles-per-second.

EXAMPLE

```
DO
  INPUT "Tone: "; Tone
  INPUT "Duration: "; Duration
  :
  SOUND Tone, Duration
  :
UNTIL (Tone=0) OR (Duration=0)
```

Example frequencies you may use in your program to create music or sound effects. (Choose the duration as required.) Quality of sound may vary by machine.

NOTES	OCTAVES						
	1	2	3	4	5	6	7
C	33	66	132	264	528	1056	2112
C^b	35	70	140	281	563	1126	2253
D	37	74	148	297	594	1188	2376
E^b	39	79	158	316	633	1267	2534
E	41	82	165	330	660	1320	2640
F	44	88	176	352	704	1408	2816
G^b	46	93	187	375	751	1502	3004
G	49	99	198	396	792	1584	3168
A^b	52	105	211	422	844	1689	3379
A	55	110	220	440	880	1760	3520
B^b	57	115	231	462	924	1848	3696
B	61	123	247	495	990	1980	3960

REMARK Some computers may not have sound. See your computer appendix for more information.



CP/M-80: Sound not supported. CHR\$(7) may sound a bell on some systems.
 TRS-80 model 1,3: Requires that a speaker be connected to the cassette port.
 TRS-80 model 4: Frequency range of internal speaker limited to 0,0 to 7,31.



See appendix for using four voice sound and utilizing the sound buffer.

SPC function

FUNCTION **SPC** (*expression*)

DEFINITION SPC prints *expression* spaces from 0 to 255

Prints the number of spaces specified by expression.

EXAMPLE DO
 PRINT "*" ; SPC(RND(20)) ; "+"
 UNTIL LEN(INKEY\$)

RUN

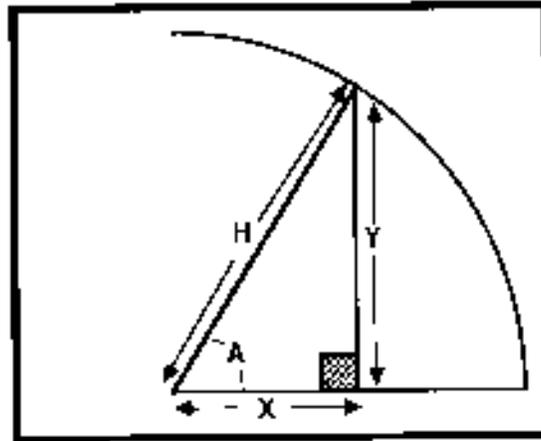
```

                +
                +
      +
            +
          +
        +
      +
    +
  +
+
```

REMARK Also see SPACE\$,LEFT\$,STRING\$,RIGHT\$,MID\$ and INSTR.

FORMAT SQR (*expression*)

DEFINITION The SQR function returns the square root of *expression*.



$$H = \text{SQR}(X^2 + Y^2)$$

$$H = \text{SQR}(X^2 + Y^2)$$

EXAMPLE A=9
PRINT SQR(A)

RUN

3

REMARK SQR is a scientific function. Scientific functions may be configured to a different precision. See "Configure" in the front of this manual for more information.

For more information on scientific functions see the "Math" and "Expression" sections of this manual and ATN, SIN, COS, TAN, EXP and ^.

STEP statement

FORMAT **FOR** *variable* = *expr1* TO *expr2* [**STEP** *expr3*]
 "
 "
 "
 NEXT [*variable*][, *variable*...]

DEFINITION This parameter allows you to set the increments used in a FOR-NEXT loop. If STEP is omitted than one is assumed.

EXAMPLE **FOR** X= 0 TO 10 **STEP** 2
 PRINT X
 NEXT
 :
 FOR X = 10 TO 0 **STEP** -1
 PRINT X
 NEXT
 END

 RUN

 0 2 4 6 8 10
 10 9 8 7 6 5 4 3 2 1 0

REMARK Also see FOR, NEXT, DO, UNTIL, WHILE, WEND and the chapter on "Loops".

 IF STEP =0 an endless loop will result.

 If *expr1* or *expr3* change while the loop is executed this change will be in effect when NEXT is encountered.

 Avoid long or complex loop expressions for *expr1* or *expr3* as they are evaluated every loop and will slow execution.

statement STOP

FORMAT **STOP**

DEFINITION **STOP** halts execution of a ZBasic program and prints the line number where execution stopped (if line numbers weren't used the lines are numbered in increments of one).

STOP when used from ZBasic will return to the Standard Line Editor.

STOP when used from a stand-alone program will return to the operating system.

EXAMPLE `PRINT "HELLO"`
`STOP`

RUN

Break in 00002
ZBasic Ready

REMARK STOP closes all files.

END may be used when no message is desired.

See TRONB and TRONX for ways of inserting break points in your programs so that <BREAK> may be used to exit a running a program.

STR\$ function

FORMAT **STR\$(expression)**

DEFINITION STR\$ returns the string equivalent of the number represented by *expression*. This is used to convert numbers or numeric variables to a string.

This function is the compliment of VAL. VAL returns the numeric value contained in a string.

EXAMPLE

```
Integer% =20000
Single! =232.123
Double# = .12323295342
:
A$=STR$(Integer%)        :PRINT A$
A$=STR$(Single!)        :PRINT A$
A$=STR$(Double#)        :PRINT A$
:
X#=VAL(A$)
PRINT X#
```

RUN

```
20000
232.123
.12323295342
.12323295342
```

REMARK Also see BIN\$, OCT\$, HEX\$, MKI\$, CVI,MKB\$, CVB and VAL.

function STRING\$

FORMAT **STRING\$** (*expr1*, *string*)
 STRING\$ (*expr1*, *expr2*)

DEFINITION Returns a string of the length *expr1* consisting of the characters specified by either the ASCII equivalent of *expr2* or the first character of *string*.

EXAMPLE PRINT STRING\$ (5, "#")
 PRINT STRING\$ (10, 65)
 PRINT STRING\$ (10, CHR\$(65))
 :
 A\$ = STRING\$(3, "*") + "TEST" + STRING\$(3, "&")
 PRINT A\$
 END

RUN

```
#####  
AAAAAAAAAA  
***TEST&&&
```

REMARK STRING\$ is more efficient than using an equivalent string of characters.

See SPACE\$, LEFT\$, RIGHT\$, MID\$, INSTR, VAL, STR\$, INDEX\$ and SPC.

SYSTEM statement

FORMAT **SYSTEM**

DEFINITION Same as END. Provided for compatibility with other versions of BASIC.

EXAMPLE PRINT"HELLO"
 SYSTEM

RUN

HELLO

REMARK Functionally identical to the ZBasic END statement. See END and STOP.



Not Supported with Apple // or Z80 versions of ZBasic. Use END.

FORMAT SWAP *var1*, *var2*

DEFINITION SWAP exchanges the contents of *var1* and *var2*. The variables can be of any type except INDEX\$ variables.

Var1 and *var2* must be of the same type.

EXAMPLE

```
B$="YES"
A$="NO"
PRINT A$, B$
SWAP A$, B$
PRINT A$, B$
PRINT
:
A=1:B=100
PRINT A,B
SWAP A,B
PRINT A,B
END
```

RUN

```
YES          NO
NO           YES

1            100
100         1
```

REMARK SWAP will execute faster and take less memory than similar methods using "holding variables".

SWAP *does not* function with INDEX\$.

TAB function

FORMAT **TAB** (*expression*)

DEFINITION Tab will move the cursor to the positions; 0 through 255, designated by expression.

Three devices may be used with Tab:

<u>DEVICE</u>	<u>FORM</u>	<u>WILL POSITION</u>
SCREEN	PRINT	CURSOR
PRINTER	LPRINT	PRINT HEAD
DISK	PRINT#	FILE POINTER

EXAMPLE

```
DATA Fred Smith, 12 E. First, Tucson, AZ, 85712
DATA Dana Andrews, 32 Main, LA, CA, 90231
:
PRINT "Name"TAB(15) "Address"TAB(30) "City"TAB(40) "State ZIP"
PRINT STRING$(50, "-")
:
FOR Item= 0 TO 1
  RESTORE Item*5
  READ N$, A$, C$, S$, Z$
  PRINT N$ TAB(15) A$ TAB(30) C$ TAB(40) S$"      "Z$
NEXT
END

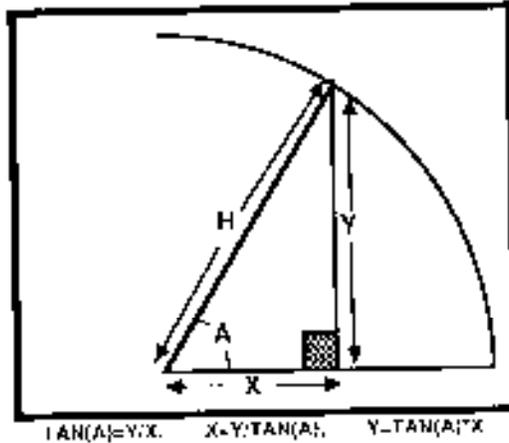
RUN
```

Name	Address	City	State	ZIP
Fred Smith	12 E. First	Tucson	AZ	85712
Dana Andrews	32 Main	LA	CA	90231

REMARK Tab will start numbering from the zero position. Also see DEFTAB,PRINT@, PRINT%,POS,PAGE,WIDTH and WIDTH LPRINT.

FORMAT TAN (*expression*)

DEFINITION Returns the value of the tangent of the expression in radians.



EXAMPLE $X\# = TAN(T+Z) / 3$

REMARK TAN is a scientific function. Scientific accuracy may be configured differently than single or double precision. See "Configure" at the beginning of this manual.

Also see ATN,COS,SIN,EXP,SQR and ^.

For more information on scientific functions see "Math" and "Expressions" in the front section of this manual.

TIME\$ function

FORMAT **TIME\$**

DEFINITION Returns an eight character string which represents the systems clock value in the format HH:MM:SS where HH=1 to 24 hours, MM= 0 to 60 minutes, SS= 0 to 60 seconds.

EXAMPLE `PRINT TIME$`
`DELAY 1000`
`A$=TIME$`
`PRINT A$`

RUN

10:23:32
10:23:33

REMARK See DATE\$ and DELAY.

This function will return a 00:00:00 if the system or version has no clock.



Macintosh: Set time from the Control Panel Desk Accessory. Also see TIMER for other ways of getting seconds.

MSDOS: Set time using TIME\$= hh, mm,ss. Also see TIMER.

Apple: See appendix for variations of system clocks.

Z80: See appendix for your particular hardware.

statement TROFF

FORMAT TROFF

DEFINITION TROFF is used to turn off the trace statements: TRON, TRONX, TRON and TRONS.

EXAMPLE

```
TRON
FOR X=1 TO 3
NEXT
:
TROFF
PRINT "Line tracing now off"
FOR X=1 TO 10
NEXT
```

RUN

```
00001 00002 00003 00002 00003 00002 00003 00004 Line tracing
now off
```

REMARK See also TRON, TRONS, TRONB, TRONX and the chapter on "Debugging Tools".

TRON statement

FORMAT TRON [{B|S|X}]

DEFINITION These statements are used for tracing program execution, single stepping through a program, and setting break points for monitoring the <BREAK> key so that you can break out of a program.

TRACING PROGRAM FLOW

TRON Prints the line numbers of the program as each line is executed so you can trace program flow and check for errors.

TRON S Lets you single step through a program. Program execution will pause at the beginning of every line in the program following TRON S (up to the end of the program or when a TROFF is encountered). Press any key to continue or press the <CTRL Z> key to enable/disable single-stepping. <BREAK> also works.

SETTING BREAK POINTS

TRON X Sets a break point at that line in a program and checks to see if the <BREAK> key has been pressed.

TRON B Sets a break point at the beginning of every line in the program following it (up to the END or until a TROFF is encountered).

Note: The <BREAK> key is checked at the beginning of a line. IF <BREAK> is encountered in a program compiled with RUN, program exits to the Standard Line Editor. If <BREAK> is encountered in a stand-alone program, exit is to the system.

<CTRL S> will pause execution when encountered during execution of TRONB, TRONX or TRON. Any key will restart. <CTRL Z> will activate/deactivate single-step mode when any TRON is active. Note: INKEY\$ may lose keys if TRON is used.

EXAMPLE

```
TRON:TRONS
```

```
:
```

```
PRINT "HELLO"
```

```
:
```

```
TROFF
```

```
RUN
```

```
00001 <KEY> 00002 <KEY> 00003 <KEY> HELLO 00004 <KEY>
```

REMARK

Every line between a TRON and TROFF may use up to eight extra bytes per line. Use TRON sparingly to save memory and increase execution speed. See chapter entitled "Debugging Tools" for more information. INKEY\$ may lose keys with TRON.



Macintosh: <BREAK> is <Command Period>. Also see BREAK ON, and TRON WINDOW in appendix for other ways of tracing program flow and variable values.

MSDOS: <BREAK> is <CTRL C>.

CP/M: <BREAK> is <CTRL C>.

Apple // ProDOS or DOS 3.3: <BREAK> is <CTRL C> or <CTRL RESET>.

TRS-80: <BREAK> is <BREAK>.

FORMAT UCASE\$ (*string*)

DEFINITION Returns a string with all characters converted to uppercase (capital letters).

EXAMPLE

```
PRINT UCASE$("hello")
:
A$="HeLLo"
PRINT UCASE(A$)
END
```

RUN

```
HELLO
HELLO
```

```
DO
  key$=UCASE$(INKEY$)
UNTIL LEN (key$)
PRINT key$
END
```

RUN

```
S          <---always returns an uppercase character
```

```
REM This function converts a string to Lowercase
:
LONG FN lcase$(string$)
  FOR X=1 TO LEN(string$)
    A=PEEK(VARPTR(string$)+X)
    IF (A>64) AND (A<91) THEN A=A+32
    POKEVARPTR(string$)+X,A
  NEXT
END FN=string$
:
PRINT FN lcase$("HELLO")
```

RUN

```
hello
```

REMARK This function is very useful when sorting data containing upper and lower case and for checking user input without regard to case.

Also see LEFT\$,RIGHT\$,MID\$,INSTR,STR\$,VAL, and the chapter "String Variables" in this manual.

UNS\$ function

FORMAT **UNS\$** (*expression*)

DEFINITION Returns a string which equals the integer value of expression in an unsigned decimal format.

EXAMPLE `PRINT UNS$ (-1)`
`PRINT UNS$ (4)`
`:`
`PRINT`
`PRINT 65535`

RUN

65535
00004

-1

REMARK This function is useful for displaying integers in an unsigned format (0 through 65,535 instead of -32,768 through 32,767).

See STR\$, DEC\$, OCT\$, HEX\$, VAL and the chapter on "Numeric Conversions".



See DEFSTR LONG for enabling this function to work with LongIntegers.

FORMAT DO
.
.
UNTIL *expression*

DEFINITION UNTIL is used to mark the end of a DO loop.
The DO loop repeats until the expression following the UNTIL is true (non-zero).

A DO loop will always execute at least once.

EXAMPLE

```
DO
  X=X+1
UNTIL x=100
PRINT X
:
"Wait for a key"
DO
  I$=INKEY$
UNTIL LEN(I$)
END
```

RUN

```
100
<KEY PRESS>
```

REMARK Notice ZBasic will automatically indent DO loop structures two spaces. See the chapter on "Formatting Program Listings" for other ways of formatting listings.

Also see FOR, NEXT, STEP, WHILE, WEND and the chapter on "Loops" in the technical section of the manual.

WHILE,WEND may be used to exit a loop immediately if a condition is false.

USR function

FORMAT **USR** *digit* (*word expression*)

DEFINITION The USR function calls the user created subroutine, defined with DEFUSR, specified by a *digit* 0 to 9, and returns the value of integer *expression* in the 16 bit accumulator.

EXAMPLE

```
REM EXAMPLE ONLY      DO NOT USE!  
:  
DEFUSR2 = LINE "Routine two"  
X=USR2(938)  
PRINT X  
END  
:  
"Routine two"  
MACHLG &8B,&C4,&C3:RETURN  
  
RUN  
  
23921
```

REMARK A machine language return is necessary at the end of a USR routine.

ZBasic provides pre-defined USR functions that perform some powerful functions like integer sine and cosine. See next page.



Macintosh: Be sure to use LongIntegers whenever referencing memory addresses. Also see CALL in the Macintosh appendix.

MSDOS: See CALL in your appendix.

Apple ProDOS: See MLI in the ProDOS appendix.

functions PRE-DEFINED USR

Predefined USR functions.

These pre-defined USR functions are available for all versions of ZBasic. See your Computer Appendix for possible other USR functions.

USR6(*expr*)

Returns the last line number executed that used any of the TRON functions (*expr* is not used).

```
TRONX
I=USR6(0)
PRINT I
```

USR7(*expr*)

Returns ZBasic's random number seed used in the RND function (*expr* is not used).

```
FOR I=1 TO 10
  PRINT USR7(0)
NEXT I
```

USR8(*angle*)

Returns the integer sine of angle in the range +-255 (corresponding to +-1). The angle must be in brads.

```
MODE7 :CLS
FOR I=0 to 255
  PLOT I<<2,-USR8(I)+384
NEXT I
```

USR9(*angle*)

Returns the integer cosine of angle in the range +-255 (corresponding to +-1). The angle must be in brads.

```
MODE7 :CLS
FOR I=0 to 255
  PLOT I<<2,-USR9*I)+384
NEXT I
```

USR statement

FORMAT **USR** *digit* (*expression*)

DEFINITION This statement will call the USR routine defined by DEFUSR digit and transfer the result of *expression* in the integer accumulator.

EXAMPLE Example only DO NOT USE
:
DEFUSR0=LINE "Machine language"
USR0(0)
END
:
"Machine Language"
MACHLG &39, &C9: RETURN

REMARK The USR routine must be set by the program or be a predefined USR routine. Also see DEFUSR, USR function, LINE, CALL, MACHLG, the chapter about "Machine Language" in this manual, and your computer appendix.



Macintosh: Be sure to use LongIntegers whenever referencing memory addresses. Also see CALL in the Macintosh appendix.

MSDOS: See CALL in your appendix.

Apple ProDOS: See MLI in the ProDOS appendix.

FORMAT VAL (*string*)

DEFINITION Returns the numeric value of the first number in a string.

The VAL function will terminate conversion at the first non-numeric character in string.

This function is the compliment of STR\$. STR\$ will convert a numeric expression to a string.

EXAMPLE

```
A$="HELLO"
B$="1234.56"
C$="99999"
:
PRINT "The value of A$=";VAL(A$)
PRINT "The value of B$=";VAL(B$)
PRINT "The value of C$=";VAL(C$)
:
PRINT
PRINT "The value of 9876.543=";VAL("9876.543")
END
```

RUN

```
The value of A$= 0
The value of B$= 1234.56
The value of C$= 99999

The value of 9876.543= 9876.543
```

REMARK The numeric value returned by VAL will be in floating point format.

See STR\$, UNS\$, HEX\$, OCT\$ and BIN\$,INT,FRAC,ABS,FIX.

Also see the chapter on "Math" and "Expressions" in the front section of this manual.

VARPTR function

FORMAT `VARPTR(variable)`

DEFINITION Returns the address of a variable . Any variable type may be used except INDEX\$.

EXAMPLE

```
A$="HELLO"  
:  
PRINT "Address of A$=";VARPTR(A$)  
PRINT "Length of A$ =" ;PEEK(VARPTR(A$))  
:  
PRINT "Contents of A4= "  
FOR X=1 TO LEN(A$)  
    PRINT CHR$(PEEK(VARPTR(A$)+X));  
NEXT  
END
```

RUN

```
Address of A$= 23456  
Length of A$ = 5  
Content of A$= HELLO
```

REMARK The following paragraphs describe which address VARPTR will be pointing to with different variable types.

INTEGER Points to the 1st byte of an integer variable.

SNG/DBL Points to the sign/exponent byte

STRING Points to the length byte

ARRAY Points to the element specified

See the sections in the front of this manual for the variable type you interested in to see how variables are stored in memory.



Macintosh: Be sure to use LongIntegers to store addresses.

MSDOS: `var=VARPTR(var)` returns two values: The address of `var` and the segment of `var` in a special variable called `VARSEG`. See appendix for details.

FORMAT **WHILE** *expression*
 .
 .
 WEND

DEFINITION This statement is used to terminate a WHILE loop. When expression becomes false the loop will exit at the first statement following the WEND.

EXAMPLE "Get a YES Answer and nothing else!"
 INPUT"What is your answer <Y/N>:";A\$
 WHILE A\$ <>"Y"
 INPUT"Please reconsider and say <Y>:";A\$
 WEND
 PRINT"Thank you for seeing things my way..."
 :
 program continues....

RUN

```
What is your answer <Y/N>: N
Please reconsider and say <Y>: Y
Thank you for seeing things my way...
```

```
WHILE X*X <23000
  PRINT X*X,
  X=X+1
WEND
END
```

RUN

```
0                   1                   4                   9                   16...
```

REMARK ZBasic will automatically indent all lines two spaces between WHILE and WEND when you use LIST. This makes programs much easier to read.

Also see FOR,NEXT,STEP,DO,UNTIL and the chapters on "Loops" and "Structure" in the front of this manual.

A structure error will occur if a WHILE exists without a matching WEND. To find a missing WEND, LIST the program and track back from the last indent.

WHILE statement

FORMAT **WHILE** *expression*
 .
 .
 WEND

DEFINITION In a WHILE statement, expression is tested for true before the loop is executed and will exit to the statement immediately following the matching WEND when expression becomes false.

EXAMPLE "GET A KEY"
 WHILE LEN(Key\$)=0
 Key\$=INKEY\$
 WEND
 PRINT Key\$
 END

RUN

<key pressed>

```
WHILE X<100
  X=X+1
WEND
PRINT X
END
```

RUN

100

REMARK ZBasic will automatically indent all lines two spaces between the WHILE and WEND when you use LIST. This makes programs much easier to read.

Also see FOR,NEXT,STEP,DO,UNTIL and the chapters on "Loops" and "Structure" in the front of this manual.

A structure error will occur if a WHILE exists without a matching WEND. To find a missing WEND,LIST the program and track back from the last indent.

statement WIDTH

FORMAT WIDTH [LPRINT][=] *byte expression*

DEFINITION Sets the allowable number of characters on a line before generating an automatic linefeed.

The optional LPRINT designates printer width.

If *byte expression* is set to 0, ZBasic will not send an automatic CR/LF. The range of *byte expression* is 0 to 255.

EXAMPLE

```
10 X=X+1
20 PRINT X
30 GOTO 10
```

WIDTH 8
LIST

```
00010 X=
      X+
      1
00020 PR
      IN
      T
      X
00030 GO
      TO
      1
      0
```

REMARK The default setting for the screen width is zero which disables the auto CR/LF after the limit has been reached.

To return WIDTH to normal, type WIDTH 79 (for 80 column screens) or WIDTH 0. When widths are set, listings are wrapped around nicely for easy reading.

To effect a smaller width, set *byte expression* to the width desired. To assure valid results for the POS statement and to keep the line position count used by tabs correct, be sure WIDTH is set to the actual screen width minus one.

WRITE# statement

FORMAT WRITE#*expr1*,{*var%*}[*var!*|*var#*]{*var\$*;*stringlength*}[,...]

DEFINITION Writes the contents of string or numeric variables in compressed format to a disk file (or other device) specified by *expr1*. The list may consist of any variable type or types, string or numeric, including arrays, in any order. *Constants may not be used!*

A string variable **must** be followed by ;*stringlength* which specifies the number of characters of that string to be written.

If the string is longer than *stringlength*, only those characters in range will be written. If the string is shorter than *stringlength*, the extra characters will be spaces.

READ# is the statement normally used to read back data written with WRITE# and will automatically read back the data written in compressed format.

EXAMPLE

```
REM The four variables below will require 18 bytes for storage
REM A$=4 bytes, A!= 4 bytes, A#=8 bytes, A%=2 bytes
:
A$="TEST": A!="12345.6":A#="12345.67898":A%=20000
:
OPEN"O",1, "DATAFILE", 18 <--- Write a file with a record length of 18
WRITE #1, A$;4, A!, A#, A%
CLOSE#1
:
OPEN"I" ,1,"DATAFILE", 18
READ#1, Z$;4, Z!, Z#, Z% <---Read in same order and type (see notes)
CLOSE# 1
:
PRINT Z$, Z!, Z#, Z%
END
```

RUN

```
TEST          12345.6          12345.67898          20000
```

REMARK

Note: Do not mix variable types when using READ# and WRITE#. READ# and WRITE# store and retrieve numeric data in a compressed format. This saves disk space and speeds program execution.

See the chapter "Files" for more detailed information using random and sequential files. Also see RECORD, LOC,REC,LOF and "Disk Error Trapping".

continued...

statement WRITE#

WRITE# continued

READ# and WRITE# STRINGS WITH VARIABLE LENGTHS

READ# and WRITE# offer some benefits over PRINT# and INPUT# in that they will read and write strings with ANY embedded ASCII or BINARY characters.

This includes quotes, commas, carriage returns, control codes or any ASCII characters in the range of 0-255.

The following programs demonstrate how to save strings in condensed format, using only the amount of storage required for each string variable.

WRITE#

```
OPEN"O",1,"NAMES"  
REM LB$=LENGTH BYTE  
DO  
  INPUT"Name: "; N$  
  INPUT"Age: "; AGE  
  LB$=CHR$(LEN(NAME$))  
  WRITE#1, LB$;1, N$;ASC(LB$), AGE  
UNTIL N$="END"  
CLOSE#1  
END
```

READ#

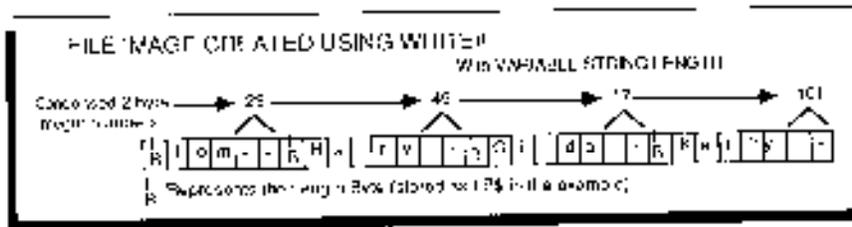
```
OPEN"I",1,"NAMES"  
REM LB$=LENGTH BYTE  
:  
DO  
  READ#1, LB$;1, B$;ASC(LB$), AGE  
  PRINT N$, "AGE"  
UNTIL N$="END"  
CLOSE#1  
END
```

The WRITE# program stores a one byte string called LB\$ (length byte). The ASCII of this string (a number from 0 to 255) tells us the length of N\$.

Notice in line 4 of READ#, that LB\$ is read BEFORE N\$, thus allowing us to read the length of N\$ first. All data in file handling statements is processed IN-ORDER.

This illustration shows how the data is saved to the disk when string data is saved using the variable length method. LB for "Tom" would be 3, LB for "Harry" would be 5, etc.

VARIABLE STRING LENGTH WRITE#



XELSE statement

FORMAT LONGIF expression
 .
 XELSE
 .
 ENDIF

DEFINITION This statement is used to separate the FALSE from the TRUE section of a LONG IF structure.

The statements following the XELSE will only be executed if the statement following the LONG IF is false.

EXAMPLE LONGIF 10 = 0
 PRINT"TRUE"
 XELSE
 PRINT"FALSE"
 ENDIF
 END

RUN

FALSE

REMARK All program lines between the LONG IF and XELSE are indented two characters when using LIST. This makes a program easier to read.

A structure error will occur the XELSE does not have a matching LONG IF.

operator XOR

FORMAT *expression₁ XOR expression₂*

DEFINITION Provides a means of doing a logical EXCLUSIVE OR on two expressions for IF-THEN testing and BINARY operations.

This operator will return true if one condition is true and one condition is false. False will be returned if both conditions are true or both conditions are false.

EXAMPLE
A\$="Hello"
IF A\$="Hello" XOR A\$="Goodbye" PRINT "YES"
IF A\$="Hello" XOR A\$="Hello" PRINT "YES"

RUN

YES

REMARK

XOR TRUTH TABLES

condition XOR condition TRUE(-1) if only one condition is TRUE, else FALSE(0)

<u>XOR</u>	<u>BOOLEAN "16 BIT" LOGIC</u>			
1 XOR 1 = 0		00000001		10000101
0 XOR 1 = 1	XOR	00001111	XOR	10000111
1 XOR 0 = 1	=	00001110	=	00000010
0 XOR 0 = 0				

FALSE XOR FALSE = FALSE
TRUE XOR FALSE = TRUE
FALSE XOR TRUE = TRUE
TRUE XOR TRUE = FALSE