K-BASIC

COMPILER

LLOYD I/O

6809 COMPUTER SOFTWARE

K-BASIC

COMPILER

K - B A S I C

BASIC COMPILER

LEVEL I

for

OS9 and FLEX

COPYRIGHT (C) 1984
by
LLOYD I/O
19535 NE GLISAN STREET
PORTLAND, OR 97230
All Rights Reserved

Contributing authors:

Frank Hoffman
Ron Anderson

* * * REGISTER YOUR COPY * * *

When you purchase K-BASIC, you purchase a single user/system license to use it. The product is supplied AS IS, however, you may receive updated versions for the operating system you use if you have registered your copy with LLOYD I/O. You will be mailed a notice of changes, new products, and other systems on which they operate.

The following update warranty is in effect. Updates are available only from Lloyd I/O unless the product was included in a system from a licensed OEM. Lloyd I/O keeps the records which prove you purchased the product. Send in your original disk (with the serial number) on which the programs came. You will receive on this disk a new copy of the current version of the program and a new manual if a new manual is needed or warranted. The cost of this update is free within 60 days of the invoice date, and $15 from 61 days and on.

Register by filling out the registration form and mail to:

LLOYD I/O
19535 NE GLISAN STREET
PORTLAND, OR 97230 U.S.A.

Software support telephone: (503) 666 - 1097

))---) MAKE SURE TO SEND IN YOUR REGISTRATION FORM (---((

NO REGISTRATION - NO SUPPORT

Copyright (c) 1984, LLOYD I/O All Rights Reserved

LEVEL I VERSION 1

JULY 1984

COPYRIGHT NOTICE

This entire manual and the associated software is copyrighted by LLOYD I/O. The reproduction of this document or associated software for any reason other than archival or backup purposes for or on the computer for which the original copy was aquired is strictly prohibited. (OS9 is a trademark of Microware Systems Corporation, P. O. Box 4865, Des Moines, Iowa 50304. FLEX is a trademark of Technical Systems Consultants, Inc. 111 Providence Road, Chapel Hill, North Carolina 27514.)

PRODUCT WARRANTY INFORMATION

The K-BASIC user manual, source code and object code software is supplied AS IS and without warranty. Reasonable care has been taken to insure that the software does function as described in this manual. If you find a situation where the program does not function as the manual describes, then contact LLOYD I/O. An attempt will be made to correct any errors brought to our attention, however we make no guarantee to do so.

IF WE DO NOT HAVE YOUR REGISTRATION IN OUR FILES
YOU WILL NOT RECEIVE ANY SUPPORT FROM LLOYD I/O.

Write, LLOYD I/O, 19535 NE GLISAN STREET, PORTLAND, OR 97230.
Phone (503)-666-1097.

DISCLAIMER

The information contained herein is believed to be accurate as of the date of publication, however, LLOYD I/O will not be liable for any damages, including indirect or consequential, resulting from the reliance upon the software or this documentation. Every attempt has been made to make this compiler compatible with existing BASIC interpreters written for FLEX. There may be some subtle differences however, and no claim is made that this compiler will generate code that will produce results identical with the previously available interpreters.

TABLE OF CONTENTS

Introduction 3
System Requirements 3
FLEX System Requirements 4
OS9 System Requirements 4
Sample Programs 4
Getting Started 5
User Prerequisites 5
Invoking the Compiler 6
Invoking the Assembler 8
Definitions 10
Compiler Fundamentals 13
Conventions 13
Program Lines 13
Expressions 15
Constants 16
Variables 17
Dimensioning 18
Assignment 19
Logical Operators 21
Mathematical Operators 21
Relational Operators 22
String Operators 22
Operator Precedence 23
Remarks 23
MEMORY (size in bytes) 25
NAME (program name text) 25
LINE=(constant byte) 26
START (address) 26
STORAGE (address) 26
DIGITS=(constant byte) 27
STRING=(constant byte) 27
PAGE 28
STTL [title text] 28
TRON 28
TTL [title text] 28
CODE 29
ENDCODE 29
TRACE 29
TROFF 29
ASSEMBLE (system command) 30
DIR (output directory),(library directory) 30
LIB (file name) 30
DATA (data)[,(more data)] 32
LET (variable name)=(expression) 32
GOSUB (label) 33
GOTO (label) 33
READ (variable list) 33
RESTORE 33
ON (word expression) GOSUB (label list) 34
ON (word expression) GOTO (label list) 34

ON ERROR GOTO ⟨label⟩ 34
RETURN 34
IF ⟨expression⟩ GOTO ⟨label⟩ 35
IF ⟨expression⟩ THEN ⟨line number⟩ 35
RESUME ⟨label⟩ 35
IF ⟨expression⟩ THEN ⟨statement 1⟩ ELSE ⟨statement 2⟩ 36
IF ⟨expression⟩ THEN ⟨statement⟩ 36
PRINT ⟨print expressions⟩ 36
INPUT [⟨literal string⟩],variable list 37
PRINT #⟨byte expression⟩,⟨print expressions⟩ 37
FOR ⟨number assignment⟩ TO ⟨expression⟩ [STEP ⟨expression⟩] 38
INPUT #⟨byte expression⟩,variable list 38
INPUT LINE #⟨byte expression⟩,string variable 38
INPUT LINE [⟨literal string⟩],string variable 38
END 39
NEXT ⟨number variable⟩ 39
STOP 39
CHAIN ⟨string expression⟩ 40
EXEC ⟨string expression⟩ 40
OPEN OLD ⟨string expresion⟩ AS ⟨byte expression⟩ 40
CLOSE ⟨byte expression⟩[,⟨byte expression⟩,...] 41
OPEN NEW ⟨string expression⟩ AS ⟨byte expression⟩ 41
KILL ⟨string expression⟩ 42
RENAME ⟨string expression 1⟩,⟨string expression 2⟩ 42
SECTOR READ,⟨drive⟩,⟨track⟩,⟨sector⟩,⟨byte array⟩ 42
CAR ⟨variable name(0[,0])⟩ 43
DIM ⟨variable name(⟨word exp.[,⟨word exp.⟩])⟩,⟨ditto⟩ 43
SECTOR WRITE,⟨drive⟩,⟨track⟩,⟨sector⟩,⟨byte array name⟩ 43
AT $⟨hex address⟩ variable 44
DPOKE ⟨word expression 1⟩,⟨word expression 2⟩ 44
POKE ⟨word expression⟩,⟨byte expression⟩ 44
REMARK ... any text ... 44
EXP(real) 46
LOG(real) 46
LOG(real) 46
ATN(real) 47
COS(real) 47
SIN(real) 47
TAN(real) 47
ABS(real) 48
INT(real) 48
RND(real) 48
SGN(real) 48
PI 49
STR$(real) 49
VAL(N$) 49
BSTR$(byte) 50
DSTR$(double) 50
LEFT$(string,byte) 50
LSTR$(long) 50
WSTR$(word) 50
CHR$(byte) 51
INCH$(byte) 51
MID$(string,byte1,byte2) 51
RIGHT$(string,byte) 51

DATE$ 52
PARAMETER$ 52
ROM$(word) 52
ASC(string) 53
BVAL(string) 53
LEN(string) 53
POS(byte) 53
CHANNEL 54
INSTR(byte,string1,string2) 54
PEEK(word) 54
WVAL(string) 54
DPEEK(word) 55
FRE(Ø) 55
HEX(string) 55
WRND(word) 55
ERL 56
ERR 56
LVAL(string) 56
PTR(variable name) 56
DVAL(string) 57
Compiler Errors 58
Assembly Errors 62
Run-Time Errors 63
Program Debugging 65
Output Files 69

##

-- Introduction --

K-BASIC is a BASIC language compiler. Level I is a simple and non-standard dialect of BASIC. It includes commands used to control the compiler, commands which are functionally translated to machine instructions, and expressions which are translated to tokens for faster evaluation.

K-BASIC actually generates assembly language source code files for assembly by the assembler. This is called a two-stage compilation. First you must write a BASIC program, compile it and then assemble it to the final executable machine code form. The only part you have to do is write the BASIC program and get the compiler and assembler to do the rest.

++

-- History --

The K-BASIC compiler design began in the spring 1982. There have been about 3000 man-hours expended in the development and testing procedures. The general idea behind the design was to create a compiler which would generate output that could be assembled by our CRASMB series of cross-assemblers. This would allow program development in BASIC for any of the CPU's supported by the CRASMB assembler. This is in fact one of the current features of K-BASIC, as it generates user definable macro calls and instructions in the assembly output files.

The compiler, assembler, and run-time software supplied are designed to generate programs that are executable on your development system.

##

-- System Requirements --

Your system will have to have disk drives and either OS9 or FLEX and about 32K or more FREE memory. The assembler requires the largest amount of memory.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

-- FLEX Systems --

The compiler is called "KS.CMD". A special text file called "KSWORDS.TXT" must exist on the same drive from which "KS.CMD" is loaded. The DIR command may be used to change this requirement. The run-time files are called "KSxxxxxx.TXT" and exist on the working drive number.

++

-- OS9 Systems --

The compiler is called "Ks" and must be copied to your execution directory (CMDS). A special text file called "kswords" must also be copied to your execution directory (CMDS). The DIR directive may be used to relocate the directory in which they are found. The run-time files are called "ksxxxxxx" and must be copied to a directory called "/D0/KS.RUN". If you are using a Radio Shack Color Computer then you may not have enough room on your drive called "/D0". If this is your case you can use your second disk drive "/D1" and use the compiler's directive DIR to change the location of the directories. Alternately you could give the run-time directory name on the OS9 command line after the file name:

```
OS9:ks prime /d1/ks.library
OS9:ks prime /d1
```

If you have purchased the software on 5.25" diskettes the software may not all fit on a single-sided single-density 5 inch disk, and therefore will be supplied on two or more diskettes. LLOYD I/O supplies only SS SD diskettes. If your system supports higher densities then you can copy the files to another diskette of higher density. Eight inch diskettes are also delivered in SS SD format.

++

-- Sample Programs --

The supplied diskette contains sample BASIC programs. For FLEX the file extension is ".BAS" and can be edited. The main file to be assembled is ".ASM" and contains all the links to the other files. The samples show the uses of variable types, functions, labels, and program statements.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

##

-- User Knowledge Prerequisites --

Users of K-BASIC should be aware that a previous knowledge of programming in the BASIC language is an absolute MUST. You should be very familiar with your operating system and your text editor. You should know the difference between source code and object code, between operating system errors and programming errors, between line labels and source code line numbers, and between a system command and a built in program's directive. There are many more things you should know which are too numerous to explain here. Using a compiler is for the experienced user, however, if you want to become an experienced user then you must get your hands dirty sometime. Now is a good time, but don't expect too much. Take a look at the 'DEFINITIONS' section for some helpful information.

++

-- Getting Started --

Probably the simplest program used by most compilers as an example is to print the message "HELLO WORLD". The following is a description of this simple program. Using your favorite editor enter the following lines into a file called "HELLO":

```
FLEX: |          TTL HELLO WORLD PROGRAM
      |          STTL SAMPLE FOR K-BASIC
      |          START 0
      |          ASSEMBLE OSM HELLO.ASM+YLS
      |
      |ALABEL    PRINT "HELLO WORLD"

OS9:  |          TTL HELLO WORLD PROGRAM
      |          STTL SAMPLE FOR K-BASIC
      |          NAME HELLO
      |          ASSEMBLE OSM HELLO.ASM Q=YLS O
      |
      |ALABEL    PRINT "HELLO WORLD"

      | <-- marks the beginning of the line (column 0)
```

Make sure you have correctly entered the program. After saving on disk then call the compiler with the following line:

```
+++KS -YLV HELLO

OS9:KS -YLV HELLO
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

If the compiler tells you it has found an error you'll have to re-edit the program and try again. The assembler will be called automatically if no errors are found. The line "ASSEMBLE OSM HELLO.ASM+YLS" makes this possible. Once the assembler is done you can try the program:

```
+++HELLO.BIN     or  +++1.HELLO.BIN
OS9:HELLO
```

The message "HELLO WORLD" should appear on your screen. This is the basic process of using the compiler. In summary there are three steps:

1. Edit a K-BASIC program.
2. Compile and Assemble to machine code.
3. Execute the final program.

Before continuing to experiment with the software you should read the rest of this manual.

##

-- Invoking the Compiler --

Making the compiler work is very simple. From the command line prompt (+++ is for FLEX, and OS9: is for OS9) type "KS" followed by a space and then the file name. This is the simplest form which will do the compile.

Example:

```
KS HELLO
```

The default FLEX file extension is ".BAS". The main output file for the assembler has an extension of ".ASM" with the same name as the ".BAS" program. OS9 uses no file extension for the input BASIC source code file.

The compiler lists the program line by line as it compiles. It finally lists the variables and labels used, the number of errors encountered, the number of variables used, the number of defined labels, the number of undefined labels, the number of referenced labels, and the number of multiply defined labels.

Now just type "KS" and hit the return or enter key. This will cause the compiler to issue its help message which will appear as:

```
Use: Ks [-opts] (file name) [(run time directory)]
  to compile K-BASIC programs
  -a don't assemble
  -b no output files generated
  -c no commented output files
```

```
-d don't delete output files
-l no program listing
-o list on pass 1 too
-p pagination mode on
-r cross reference labels
-s no label table listing
-v no variable table listing
-x cross reference variables
-y yes delete old output files
-? display this menu
```

K-BASIC will display this message when it detects errors in operating procedures that are fatal. If you can't remember the options that may be used with K-BASIC then typing "Ks" and return or enter will help you.

Here is a complete description of each option:

-a by default K-BASIC will call the assembler if no errors were detected, and the ASSEMBLE (parameters) directive was used in the source code except when the '-a' option is used.

-b by default K-BASIC will generate output code except when the '-b' option is used. This option MUST be used when the '-r' or '-x' cross referencing options are used.

-c by default K-BASIC will use the input source code lines as comments in the output code except when this option is used.

-d by default K-BASIC will generate the code to allow the OSM assembler to delete the intermediate assembly code files except when this option is used. All but the ".ASM" file will be deleted.

-l by default K-BASIC will list the input source code file while it is being processed during pass 2 except when this option is being used.

-o K-BASIC will list the input source code file during pass 1 if this option is used and the '-l' option is NOT used. This is a compiler debugging feature and is normally not used.

-p by default K-BASIC does not paginate except when this option is used. The pagination mode allows form-feeding across the bottom of the page, page numbering, and page titling.

-r by default K-BASIC does NOT cross reference labels unless this option is used in combination with the '-b' option. Cross referencing provides information for debugging and making your software more efficient.

-s by default K-BASIC lists the labels used in a program except when this option is used. This option must not be used when a cross referenced label list is to be produced.

-v by default K-BASIC lists the variables used in a program except when this option is used. This option must not be used when a cross referenced variable list is to be produced.

-x by default K-BASIC does NOT cross reference variables unless this option is used in combination with the '-b' option.

-y by default K-BASIC will ask if you want to delete any existing output files unless this option is used. Then K-BASIC will automatically delete them.

-? K-BASIC has been designed to list its help message when errors have been detected or if this option has been used.

The TTL and STTL directives can be used to set up the page titles. The PAGE directive causes an eject to the next page. While in the pagination mode the compiler numbers each page prints the TTL and STTL titles, the date and version number of the compiler.

Example:

```
FLEX:    KS -YLVS HELLO
OS9:     KS -YLVS HELLO
```

This example will auto-delete the output files, suppress the program, label table, and variable table listings, and finally print the ending report.

Example:

```
FLEX:    KS -YP HELLO
OS9:     KS -YP HELLO
```

Here the output files will be auto-deleted and the pagination mode will be turned on. The program will be listed as it is compiled and the variable table and label table will be listed.

++

-- Invoking the Assembler --

K-BASIC can call the assembler by the use of the ASSEMBLE directive. This is a conditional command in that it doesn't immediately do any thing other than save the text following the word ASSEMBLE into a buffer. When the compiler is done it checks the error counter for a zero value. If it is zero (meaning no errors were found) or the command line option '-a' was NOT used the compiler feeds the text from the ASSEMBLE directive buffer to the operating system for execution.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

The compiler generates several assembly language text files. The main one to be assembled is named the same as the input basic file, but with an extension of ".ASM". This means the assembler has to use a file with an extension of ".ASM".

Example:

```
FLEX:    ASSEMBLE OSM HELLO.ASM+YLS
OS9:     ASSEMBLE OSM HELLO.ASM Q=YLS O
```

For more details on invoking the assembler see the assembler manual.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

##

-- Definitions --

The following table of definitions explains the use of things (syntax, formats, ...etc).

Word or form	definition
[item]	the 'item' is optional
(item)	the 'item' is required
[(item)]	the 'item' is optional but must conform to a certain type of syntax.
address	a constant hexadecimal number. A word number in the range of $0000 to $FFFF.
assignment	moving data to a variable in the right format for the variable type.
byte	a byte integer expression (8 bit signed)
compile time	when a BASIC program is being compiled to assembly code and when the assembly code is assembled to machine code.
constant	a literal string or number which is converted to its natural form: byte -1 to $FF word 10 to $000A
control	control types of statements are GOTO, GOSUB, RETURN, ON ERROR GOTO, RESUME, ON A% GOTO, ON A% GOSUB, the conditional IF, and the loop control FOR-NEXT statements. Each of these control the next statement to be executed in the run-time program.
CR	a carriage return (or 'enter' key)
CRLF	outputs a Carriage Return and Line Feed pair of characters to cause a new line to be started.
dimensioning	the process of setting subscript limits and allocating memory for the data storage of the newly created variables.
ND	non-dimensioned VDT
SD	single dimensioned VDT(1)

DD	double dimensioned VDT(1,1)
directive	a command immediately executed during the compilation process
double	a double integer expression (64 bit signed)
expression	a mathematical phrase consisting of operands (numbers, variables, and functions) and operators (+ - *... etc)
function	an operand that is not a constant or a variable, but changes the value of an operand to another type or transforms it to another value of the same type based upon the value passed to it.
intrinsic	belonging to BASIC, built into the compiler and run-time package.
label	a character sequence beginning in column one of a program line which must be unique. It may also be used by program control transfer statements like GOTO or GOSUB.
line	a program line containing an optional label and statement or directive. The three types: 1. [(a blank line)] 2. [(*) a comment line] 3. [(label)] (directive or statement)
line number	a label used for program control transfer statements like GOTO and GOSUB.
long	a long integer expression (32 bit signed)
operands	a number, variable name, or function used in an expression.
operator	a sequence of characters dividing two or more operands in an expression that defines how the two operands are to be combined into a single operand.
option	a choice selected by a directive or command line switch.
precedence	the order of operations, i.e. multiplication and division are done before addition and subtraction. Follows the rules of algebra.
pointer	a variable that is used to contain the address of another piece of data.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

real — a real number expression; +/-9.999999999999999E+/-99 or 15 digits (9 bytes per real number)

run time — when the machine code program is loaded into memory and is executing.

statement — an instruction that is translated to machine code for later execution; one of BASIC's key words with its proper syntax.

string — a string expression (up to 255 characters)

subscript — the expression used in a dimensioned variable as an indirect pointer to the actual variable data.

syntax — the arrangement of the words and expressions in a directive or statement in their proper forms and relations.

termination — a way of ending or stopping further processing or execution.

terminator — a character that begins and ends a series of characters: 'hello' is a literal string string terminated with a single quote (').

type suffix — the suffix character used to define the type of variable used in an expression or assignment:

real	no suffix	ANGLE
string	$	Name$
byte	'	count'
word	%	lipe%
long	#	dollar#
double	~	Mtotal~

variable — a named location in memory containing the value of a real, string, byte, word, long, or double number; the name may be used in another expression representing the value.

word — a word integer expression (16 bit signed)

Copyright (c) 1984, LLOYD I/O All Rights Reserved

##

-- Conventions --

A program to be compiled by K-BASIC consists of a text file containing lines of basic code which you have typed using your editor. Each line must consist of the ASCII characters $20 to $7F followed by a CR ($0D). This is the normal format that most editors create.

The allowable characters are:

```
 !"\#$\%&'()*+,-./0123456789:;<=)?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~)
```

To make it easier to explain the definitions of fundamentals, directives, statements, and functions, items enclosed within angle brackets (<>) are essential elements which must use a certain syntax, and items enclosed within the square brackets ([]) are optional. Items enclosed within angle and square brackets are optional but if it is used then it must use a certain type of syntax.

Examples:

```
<real>  <label>          required
[-opts]                  optional
[STEP <expression>]      optional expression
```

##

-- Compiler Fundamentals --

++

Program Lines

K-BASIC accepts lines generated by any text editor, however, control codes are not allowed (character values less than 32). The maximum length is 127 characters and a CR (carriage return).

The general formats are:

1. [< a blank line>]
2. [* a comment line for documenting purposes]
3. [<label>] <directive or statement>

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Format one is a blank line. It is used for spacing out lines for better documentation of the program, separating subroutines from each other...etc.

Format two is a comment line. The line begins with an asterisk and is not processed for directives or statements. When you are writing and debugging a program, this is an easy way to exclude portions of code from the compilation.

Format three is the main line which is processed by the compiler. The label begins in column one and ends with a space. If a label is not used then the line must begin with a space.

```
COLUMN            1         2         3         4         5
NUMBERS 12345678901234567890123456789012345678901234567890123 4

                  TTL PROGRAM TO FIND X DATES
                  STTL (c) 1984 by LLOYD I/O
                  START 0
        MAINLOOP  PRINT
                  FOR X%=1 TO 10
                   FOR Y%=X%+1 TO 11
                    IF ARRAY%(X%))ARRAY%(Y%) THEN GOSUB SWAPDATA
                   NEXT Y%
                  NEXT X%
                  END

        *    SUBROUTINE TO SWAP DATA

                  SWAPDATA LET B%=ARRAY%(X%)
                  LET ARRAY%(X%)=ARRAY%(Y%)
                  LET ARRAY%(Y%)=B%
                  RETURN
```

Labels can be the traditional line numbers used by most BASIC interpreters. Labels can also be just a series of up to 16 characters, all of which are significant. K-BASIC only uses labels which have been referenced by a GOTO, GOSUB,... etc, and converts the name to a shorter form to conserve memory during the assembly process.

Examples:

```
100
110
120
130
LOOP
AVERYLONGLABEL
ASHORTERONE
S
```

Labels may consist of any series of characters except for a comma (,) or a space ( ). There is one exception to this rule. The conditional statement "IF" must use line numbers when a

conditional branch is to be made.

Example:

```
IF A=0 GOTO 100
IF B$='END' THEN 4000 ELSE 2000
```

Upper case or lower case is acceptable, however, traditional labels and variable names should be in LOWER case, key words for directives, statements, and functions should be in UPPER case.

Very long lines can be concatenated by ending the line with a backslash (\). K-BASIC checks each line as it is read in for the backslash. If found, the next line will be appended to the end of the current line. This is a way to input very long lines. The absolute maximum length is 256 characters.

Example:

```
          IF QUESTION$(XREGISTER%,YREGISTER%)=1 AND QUESTION$\
(IREGISTER%,JREGISTER%)=1 THEN GOSUB 400 ELSE GOSUB 5000
```

In order for the expression in the conditional to be correctly evaluated by the compiler, the second line must begin with the next part of the expression.

++

Expressions

Expressions evaluated for later use in the running program are converted to a "tokenized" form. The operands are made up of constants and variables which are explained later. The operators consist of the mathematical, logical, and relational operatives. An expression can be as simple as one operand or any number of operands separated by any of the operators.

In general there will be one less operator than operands in any given expression. The operators follow the algebraric rules of operator precedence. There are some rules that must be followed. Except for the use of some of the functions, expressions cannot mix the various types of constants and variables. I.E. you cannot assign a real number to a string, or a double integer to a real, or a byte to a long integer...etc. There are six types available:

1. real numbers...15 digits with an exp. of +/- 99
2. character strings...up to 255 characters
3. signed 8 bit byte integer numbers
4. signed 16 bit word integer numbers
5. signed 32 bit long word integer numbers
6. signed 64 bit double long word integer numbers

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Real numbers are stored in 9 bytes in BCD format. The ninth byte is the 2's complement exponent. The first byte has the first digit in the lower four bits, while the sign of the number is the 7th bit. Bits 4, 5, and 6 are unused, and should be left so. The range of values is from +/-9.99999999999999E-99 to +/-9.99999999999999E+99.

Character strings are formated as follows. The first byte is the length of the string. If the value is zero, then the string is said to be a null string. The string can contain up to 255 characters ($00 to $FF). The "STRING=" directive sets the maximum length processed by the run-time package and it defaults to 64 characters. Each character of a string can be any value from $00 to $FF.

Byte numbers are signed and occupy only one byte of memory. Word numbers take up two bytes of memory. Long word numbers take up four bytes of memory. Double long word numbers take up eight bytes of memory.

The range of values for byte numbers is -128 to +127. The range of values for word numbers is -32768 to +32767. The range of values for long word numbers is -2,147,483,648 to +2,147,483,647. The range of values for double long word numbers is -36,028,797,018,963,968 to +36,028,797,018,963,967.

++

Constants

All constants are converted to their natural form and are written out to a file with an extension of ".CON" ready to be assembled. Here are some examples of number constants.

Examples:

2.739	a constant real number
$E090	a constant word number
5	a constant integer
-45	ditto
-4.009E33	a constant real number
1e-80	ditto
-$4001	a constant integer

String constants (literal strings) may be terminated with either single (') or double (") quotes. They may also contain a colon (:). The following are some string constants.

Examples:

"Hello world"	a constant string
'What is your name please'	single quotes as terminators
"Report to me now!"	double quotes as terminators
'Input a key :'	a string with a colon

```
'the word "HELP" is for'     a string with double quotes
""                           a null string
```

++

Variables

There are 6 major types of variables with 6 sub-divisions each for a total of 36 types. For each major type (real, string, byte, word, long, and double) there are the following 6 sub-divisions:

1. non-subscripted
2. single dimensioned
3. double dimensioned
4. AT defined non-subscripted
5. AT defined single dimensioned
6. AT defined double dimensioned

The "AT" statement defines an absolute address of where the data for a variable is to be stored.

Each variable is given a name starting with a letter (A-Z or a-z) and can consist of letters or numerals to a maximum length of twelve (12) characters.

Examples:

```
A
B
H1
HELLOMESSAGE
SIZZLER
SNAKE
CORN1
CORN2
```

These are all legal variable names. The way K-BASIC tells the difference between variables of the same name is by a suffix character. The following table defines these suffixes:

character	variable type	bytes of memory
none	real	9 bytes
$	string	1+length
'	byte	1 byte
%	word	2 bytes
#	long word	4 bytes
~	double long word	8 bytes

It is also possible to have variables of the same type and name, but differing in their subscript size (none, single, or double), however, absolute variables come from the same definition as the normal variables, so it is impossible to have two variables of the same type and subscript size, one being a normal and the other an absolute.

Remember "ND" means non-dimensioned, "SD" means single dimensioned, and "DD" means double dimensioned. The following are some examples of legal variables.

Examples:

```
realcount          a ND real variable
ANGLE              a ND real
ANGLE(0)           a SD real
Taster(0,0)        a DD real

HELPMESSAGE$       a ND string variable
name$              a ND string
questions$(31)     a SD string
an$(10,10)         a DD string

COL'               a ND byte integer variable
Counter'           a ND byte
AA'(49)            a SD byte
video(25,51)       a DD byte

CHANNELS%          a ND word integer variable
cal%(9)            a SD word
AM%(51,20)         a DD word

dollars#           a ND long word integer variable
ARRAY#(500)        a SD long
Wholen#(10,10)     a DD long

Bignames~          a ND double long word integer
Totald~(200)       a SD double
tt~(31,4)          a DD double
```

Variables are defined as they appear in the program, and are listed in that order at the end of the compile. They are allocated storage in the file with an extension of ".RMB". Each variable is converted to a short form representation to conserve memory during the assembly process.

++

Dimensioning

All six variable types can be dimensioned with either one or two subscripts. The basic rule to remember is that all subscript values must be represented with word integer expressions. A variable is dimensioned by either the "DIM" or "AT" statement followed by a word integer expression enclosed in parentheses or two word integer expressions that are enclosed in parentheses but separated by a comma. The word integer expression is called the subscript. Once an array (a dimensioned variable) is defined it can be closed and re-dimensioned with different values for the subscripts. The "CAR" statement closes an array.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Some examples will help clarify what has just been described.

A single dimensioned array: DIM AB(3)

creates four new variables: AB(Ø), AB(1), AB(2), AB(3)

A double dimensioned array: DIM AM(1,3)

creates eight new variables: AM(Ø,Ø), AM(Ø,1), AM(Ø,2), AM(Ø,3)
AM(1,Ø), AM(1,1), AM(1,2), AM(1,3)

In either case word integer expressions can be used as subscripts and the array itself can be another part of an expression.

Example:

```
AM(XREG%(X%),YREG%(Y%))
```

++

Assignment

Variables can be assigned values from the LET, INPUT, and the READ and DATA statements.

The LET statement assigns the value of an expression to a variable. The variable can be non, single, or double dimensioned, normal or absolute; of the six types real, string, byte, word, long, or double.

Examples:

```
LET A=(PI/180)*Angle
LET B~(X%)=1+(405/EDGE~)
LET NAMEQ$="Enter your name "
```

The word "LET" is optional in most statements, however, in most cases of the conditional "IF" statement it should be used. Notice the form:

```
variable name=expression
```

The variable name is always on the left side of the equals sign ("=") and the expression is on the right side. This follows the normal syntax for all BASIC compilers and interpreters. However, some BASIC's allow embedded spaces within the phrase. This is NOT allowed in K-BASIC. The reason is not memory savings (because once compiled spaces aren't saved anyway), but rather that it would make the compiler's syntactical analyzer much more complex. Therefore, the simple rule: no embedded spaces in any expression. (Except of course within literal (constant) strings.)

Copyright (c) 1984, LLOYD I/O All Rights Reserved

The INPUT statement gets a string of characters from the standard input device or from a file and converts it to the internal format of the type of the variable it is assigned to as given in the variable list following the word 'INPUT'.

Examples:

```
INPUT A              input a real number
INPUT NAME$          input a character string
INPUT CHO'           input a byte integer
INPUT WET%           input a word integer
INPUT count#         input a long integer
INPUT PEOPLE~        input a double integer
INPUT A(EUI%)        input a single subscripted real
```

In general the INPUT statement allows interactive input into a program from the terminal, or the reading of text files from disk. To input from disk the channel number must be given:

Examples:

```
INPUT #1,A           input a real number from channel 1
INPUT #9,NAME$       input a string from channel 9
INPUT #CH',A$(X%)    input a SD string
INPUT #1,A,B,C       input three real numbers
```

The variable list can be one or more variables, and is limited in the number of variables only by the size of the BASIC program line. The channel numbers range from 0 to 12, channel zero being standard input, and the disk channels are from 1 to 12. Of course in OS9 an input path can be opened from almost any device.

The last type of assignment is with the use of the READ and DATA statements. The DATA statement defines a series of one or more constants all separated by a comma (,). The constants are written out to a file with an extension of ".COD". A pointer is set to the beginning of this 'string' of constants. Then each time the READ (variable list) statement is used the INPUT statement line pointers are set to the DATA string pointers. This allows any of the INPUT variable assignment types to be used.

Examples:

```
100       READ NAME$(year%),AGE
          ...
900000    DATA ADAM,899
          DATA NOAH,450
          DATA MOSES,120
```

Any amount of DATA statements is allowed. Each time the READ statement is executed a new value is assigned to the variables in its list. Therefore the READ DATA statements can be used to set up arrays, define dictionaries, ...etc.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

Mathematical Operators

The operators available are:

^	exponentiation	X^Y
*	multiplication	X*Y
/	division	X/Y
+	addition	X+Y
-	subtraction	X-Y

These are the operators most people are familiar with in their exposure to most BASIC interpreters. All the integer operations are done in signed arithmetic. However, integer exponentiation has some limits:

1. the exponent is limited to a positive number
2. the exponent is limited to a value between 0 and 255.

Real number exponentiation is allowed with negative and fractional exponents.

++

Logical Operators

The logical operators available are:

<<	shift left	X'<<Y'
>>	shift right	X'>>Y'
COM	complement	COM X'
AND	and	X' AND Y'
OR	or	X' OR Y'
EOR	exclusive or	X' EOR Y'

The shift operators shift the operand on the left bit by bit by the 'count' value of the operand on the right.

Examples:

5<<2	becomes 20
80<<4	becomes 1280

The gate logic operators COM, AND, OR, and EOR have different meanings for real, string, and integer types of operands. Only the first byte of a real number is affected. The COM of a real number is the 9's complement. The COM of an integer is the 1's complement. The COM of a string is not allowed. Only the length byte of strings is affected. All bits of the four integers (byte, word, long, double) are affected.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

When the logical operators are used with real numbers and character strings, they are intended to be used with the relational operators:

Example:

```
IF (AN$="YES") AND (QAN$(X%)="YES") THEN GOSUB 980
```

The logical operators COM, AND, OR, and EOR must have a space on either side. One and only one space is allowed and it must be there. The space is used to allow variables that may have key-words used as a part of their name.

++

Relational Operators

The available relational operators are:

<>	><	not equal	X<>Y
<=	=<	less than or equal	X<=Y
>=	=>	greater than or equal	X>=Y
=		equal	X=Y
<		less than	X<Y
>		greater than	X>Y

The relational operators compare the operand on the right to the operand on the left. If the comparison is true a value of one (1) is returned. If it is false a value of zero (Ø) is returned. When strings are compared a null string is returned on a false condition and a string with a length of one (1) is returned on a true condition. A true string comparison returns the same string as a CHR$(Ø). A false string comparison returns the same string as a "" (a null string).

The use of a one (1) as true and zero (Ø) as false allows the correct usage of the logical operators AND, OR, and EOR when used with the IF statement.

++

String Operators.

The string operators allowed are +, which concatenates, joins, or splices together, the two strings; all six relational operators; and the AND, OR and EOR logical operators. The relational operators do an alphabetical comparison between the two strings. A null string is always less than a string of a length greater than zero. Two null strings are equal.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

Operator Precedence

The operator precedence follows the rules of algebra. In the following list the operator at the top of the list has the highest precedence, while the one at the bottom has the lowest. Operators of equal precedence are evaluated from left to right in the expression.

1.	()	expressions enclosed in parentheses
2.	-	unary minus
3.	COM	the logical complement operator
4.	^	exponentiation
5.	* /	multiplication and division
6.	+ -	addition and subtraction
7.	<> <= >= = < >	relational operators
8.	AND OR EOR	the logical AND, OR, and EOR operators
9.	<< >>	the logical shift left and right operators

The following table describes which operators are allowed with the different data types:

operator	real	string	integer
()	yes	yes	yes
-	yes	no	yes
COM	yes	no	yes
^	yes	no	yes
*	yes	no	yes
/	yes	no	yes
+	yes	yes	yes
-	yes	no	yes
<> ><	yes	yes	yes
<= =<	yes	yes	yes
>= =>	yes	yes	yes
=	yes	yes	yes
<	yes	yes	yes
>	yes	yes	yes
AND	yes	yes	yes
OR	yes	yes	yes
EOR	yes	yes	yes
<<	no	no	yes
>>	no	no	yes

++

Remarks

The REM statement is the traditional way of including comments in a program. In K-BASIC any blank line (a line consisting of just a CR) is considered to be a remark in that it spaces out the program, dividing up sections for easier readability. A line beginning with an asterisk ("*") is also a comment.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

It is good programming practice to use remarks or comments freely throughout your programs. All BASIC input program lines are written out to the main assembly program file with an extension of ".SUB" making the output program easy to read. However, REM or REMARK do not use up any run-time memory because remarks are not needed for any execution purposes.

It is recommended, however, that you use the "*" form for most comments and remarks.

Examples:

```
* this subroutine has the following functions
*   1. inputs an address from the terminal
*   2. allows the address to be edited before returning
*   3. puts the address into the array "ADDRESS$(5)"
*   4. is set up for non-DEC terminals, ie terminals
*      using the simple ESC-code sequences.
```

It is a good practice to head up all subroutines with a description of their function, what is expected for input, and what is delivered as output.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

##

-- Compiler Directives --

++

Program Location

NAME <program name text>

The OS-9 version of K-BASIC requires that the program module generated be given a name. The name cannot be any longer than 29 characters. The name can not have any embedded spaces. The FLEX version ignors the name directive.

Examples:

right	wrong
NAME HELLO	NAME HELLO THERE
NAME Stampaddress	name Stamp Address
name sortm	NAME Sort merge it

All three wrong lines are in error because no imbedded spaces are allowed in the program name.

MEMORY <size in bytes>

This directive is only used in the OS9 version for setting the amount of memory to be used for general purpose use such as string and array storage. In addition to this about 4k more will be assigned for the stack. The remaining a section of memory used by programs is used for pointers and system variables and buffers. If this directive is not used a default size of 8k will be assigned. When designing your software you are responsible for making sure enough memory is assigned.

Examples:

right	wrong
MEMORY 8000	MEMORY TOTAL
MEMORY $4000	MEMORY A000

Wrong line one is in error because variables can not be used. The same is true of the second line, however, it appears that the '$' is missing from the 'A000' which should go like '$A000'.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

START <address>

The <address> must be a constant expression using integer values. As supplied K-BASIC does not use this directive. The user must modify the run-time files to make use of it. K-BASIC is set up to have an starting address of $0000.

Examples:

right	wrong
START $C100	START C100
START 0	START ZERO
START $0000	START 0+OFFSET

Line one is wrong because C100 should be $C100. Variable names are not allowed as in lines two and three.

STORAGE <address>

The address is the base address of the storage. The 6809 run time package does not use the data generated by this command. Again it is up to the user to modify the run-time package to use it. The FLEX version allocates user data storage at the end of the program to the end of user memory. The OS9 version is assigned user memory when the program is started. The size may be varied by the MEMORY directive.

Examples:

right	wrong
STORAGE 0	STORAGE BEGIN
STORAGE $B000	STORAGE B000

Line one is wrong because again variables are not allowed and line two is wrong because B000 is a hex number and requires the dollar sign ($B000).

++

Special Length Control

LINE=<constant byte>

The length of an output line will be automatically restricted in length to the value specified in the assignment.

Examples:

right	wrong
LINE=0	LINE=ZERO
LINE=80	LINE=80-9
LINE=65	LINE=65+OFFSET

The first wrong line is in error because it contains a variable. The second two are in error because they are expressions. The last line is also wrong because it contains a variable.

STRING=<constant byte>

The mathematical processor for strings is limited to the string length specified. The default is 64 characters. You should set it for the maximum length of any string you might expect.

Examples:

right	wrong
STRING=32	STRING=Slen
STRING=255	STRING=0

Again, variables are not allowed and the minimum value is limited to one. The first wrong line contains a variable and the second contains a zero.

DIGITS=<constant byte>

The digits printed after the decimal point on a floating point number are restricted to the value given. The range is limited from zero to fifteen (0-15).

Examples:

right	wrong
DIGITS=2	DIGITS=30
DIGITS=15	DIGITS=RLEN
DIGITS=0	DIGITS=16

The first and third wrong examples are in error because the value given is out of range. The second wrong example is in error because it uses a variable.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

Pagination

TTL [title text]

This directive sets the main title for the compiler and the assembler.

Example:

```
TTL       Program to print "hello world"
```

STTL [title text]

This directive sets the sub-title for the compiler and the assembler.

Example:

```
STTL      (c) 1984 by LLOYD I/O
```

PAGE

This directive sends a form feed if, and only if, the "P" option switch was used on the system command line in FLEX or OS-9. A new title will be generated using the TTL and STTL titles, page number, and date.

Example:

```
PAGE
```

++

Debugging Aids

TRON

This directive causes support subroutines to be included which, when used with the TRACE directive,can be used to display the value of the current program counter and source code line number. It should also be used if the ERL (error line) is to be used. It causes the source code line number for each line compiled to be set into the variable ERL before the rest of the line is executed at run time. This continues until the TROFF directive is used.

Example:

```
TRON
```

TROFF

Turn off the trace mode. If TRON or TRACE was used then ERL will remain with the line number of the TROFF directive.

Example:

```
TROFF
```

TRACE

This directive will turn on the trace mode and print a message similar to:

```
TRACE MODE AT $xxxx IN LINE # n
```

where '$xxxx' is the hex address of the next source code instruction and 'n' is the source code line number. TRACE also turns on the trace mode.

Example:

```
PRINT "HELLO WORLD"
TRACE
PRINT
```

++

Assembly Code Passing

CODE

Everything between the CODE and ENDCODE directives is passed to the output file for assembly. It goes to a file with an extension of ".SUB".

ENDCODE

This is the closing directive for the CODE directive.

Examples:

```
CODE
LEAX      49,U
LBSR      INTSEX
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

```
JSR       PCRLF
ENDCODE
```

++

Miscellaneous

LIB (file or path name)

This directive allows programs to call other BASIC routines for inclusion in the program. This is an effective way to control program development and maintenance. It is not nestable.

Examples:

	right	wrong
FLEX:	LIB INPART1	LIB INPARTONE
OS9:	LIB Inpart1	LIB 1.INPART1

The FLEX wrong line is in error because the file name is too long, but it would be allowable in OS9 because file names are limited to 29 characters. The OS9 wrong line is in error because it uses a FLEX file specification which is not compatible with OS9 path names.

ASSEMBLE (system command)

This command sets up the (system command) as a string which is passed to the system for execution if the program being compiled has no errors. This is an effective way to call the assembler when the compilation is complete.

Examples:

```
FLEX:     ASSEMBLE OSM HELLO.ASM+YLS
FLEX:     ASSEMBLE EXEC ASS.INVENT

OS9:      ASSEMBLE OSM HELLO.ASM Q=YLS O=HELLO
OS9:      ASSEMBLE ASS.INVENT
```

DIR (Output files directory),(Run time files directory)

The DIR command tells the compiler which tells the assembler where the output and the run time files are located. By default the current working drive or directory is used for the output files. The FLEX version also assumes the working drive for the run time files. The OS9 version assumes the run time files are in a directory called "/D0/KS.RUN". DIR can change these

default drive numbers or directories.

Examples:

```
FLEX:    DIR 1,0
OS9:     DIR D1/ACC1,/H0/BASIC
```

The assembler directive CHD (drive or path) is used to change the current working drive number or directory on a temporary basis. When the assembler has completed it returns the user to the same drive or directory as before the assembly.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

##

-- Run-Time Statements --

++

Assignment

LET ⟨variable name⟩=⟨expression⟩

⟨variable name⟩=⟨expression⟩

This is the general form of assignment. The expression type must match the variable type. The compiler will automatically generate an assignment statement when it finds that no key word matches the variable name.

Examples:

```
right                          wrong
LET A=9.9                      LET A = 9.9
LET CUSTNAME$=LEFT$(N$,9)      LET CUSTOMERSNAME$=N$
A'(X%)=PEEK($CC01)             A'(X)=PEEK($CC01)
ENDMEM%=DPEEK($CC2B)           ENDMEM=DPEEK(CC2B)
```

The first wrong line has embedded spaces in the assignment expression. The variable name in line two is too long. The subscript in line three should be a word type instead of the real type. The 'ENDMEM' variable in line four is real, while the expression is of the word type. This is a mis-match assignment error.

DATA ⟨data⟩[,⟨more data⟩]

The DATA statement reserves the data for later assignment by the READ statement. Data is stored in one file called ".COD".

Examples:

```
DATA 1,9,20,0,9,2,-3
DATA FRANK,1,27,56,7,11,-1
DATA LEWIS,3,33,61,3,1,-1
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

READ <variable list>

This statement reads data into the variables in much the same way as the INPUT statement evaluates string input and converts it to the form specified by the variable type.

Examples:

```
READ A,BET(H%),AGE,EF,TOOOLD,COUNT,EOF
READ NAME$,DEPENDENTS',AGE',YY',MM',DD',EOF'
```

RESTORE

The current DATA pointer is reset to the beginning of the data. The data is defined in the file ".COD".

Example:

```
RESTORE
```

++

Transfer and Control

GOTO <label>

Program execution is transfered to the location. The label can be a line number or a standard label.

Examples:

```
         GOTO 100
         GOTO MAINLOOP
         GOTO Catch22
LOOPFOR  GOTO LOOPFOR                    (loop forever)
```

GOSUB <label>

Program execution is transferred to the location until a RETURN statement is found at which point control is transferred to the statement following the GOSUB statement.

Examples:

```
         GOSUB 8000
         GOSUB PRINTDATA
         GOSUB INPUTADD
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

RETURN

Program execution is returned to the statement following the GOSUB statement. This is the normal way of exiting a subroutine. The return statement checks to see if it is permissiable to return, as a return can only be done if a GOSUB or ON <> GOSUB was executed.

Example:

```
RETURN
```

ON <word expression> GOTO <label list>

This is a multi-way branch statement based upon a positive expression value which indexes a lookup table of label addresses. This is an effective way of multiple choice branching.

Examples:

```
right                          wrong
ON SELECT% GOTO 100,200,300    ON B GOTO 100,200,300
ON TEMP% GOTO LOW,HIGH         ON COMP' GOTO LIST,DUMP
```

The two wrong lines are in error because the wrong type of expressions have been used. Only word expressions are allowed.

ON <word expression> GOSUB <label list>

This statement is the same as the ON GOTO statement except that a branch-to-subroutine is done.

Examples:

```
right                          wrong
ON SELECT% GOSUB 100,200,300   ON B GOSUB 100,200,300
ON TEMP% GOSUB LOW,HIGH        ON COMP' GOSUB LIST,DUMP
```

Again the two wrong examples are not using word expressions. NOTE: The range of the word expression is limited in value from one (1) to the number of labels. A value exceeding of this range results in error number 32.

ON ERROR GOTO <label>

Error trapping is set to the label. When the run time package detects an error it will branch to the label and continue execution from there. At that point the ERR variable should be checked for valid errors. To change back to the built-in error reporting system use "ON

ERROR GOTO"

Examples:

```
ON ERROR GOTO ENDOFFILE
ON ERROR GOTO ERRORDET
ON ERROR GOTO
```

RESUME <label>

After an error has been handled control should be passed to the main program flow with this directive because it sets up the system stack pointers. However, do not RESUME to a place where a RETURN statement will be executed because your program will go off into the never-never land of no-return.

Examples:

```
RESUME    RECOVERED
RESUME    MAINLOOP
```

++

Conditional Program Transfer and Control

IF <expression> GOTO <label>

If the value of the expression is NOT EQUAL TO ZERO then program control is transferred to the address of the label.

Examples:

```
IF IN$="END" GOTO ENDPROGRAM
IF A'<9 GOTO RETRY
IF ABS(RZ)<=4E9 GOTO THROUGH
```

IF <expression> THEN <line number>

This does the same as the IF GOTO statement, however, the 'line number' (label) is limited to line numbers.

Examples:

```
IF X=1 THEN 100
IF A$="" THEN 52900
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

IF (expression) THEN (statement)

With this form the statement is executed if the expression is true. Any valid statement and assignments are allowed.

Examples:

```
IF CC'<>1 THEN PRINT "CALCULATOR IS OUT OF SYNC"
IF DO$="CAT" THEN EXEC "CAT"
```

NOTE: The compiler tells the difference between labels and statements used in a conditional clause by limiting labels to line numbers. This is because assignments and statements use the same start as alphabetical labels.

IF (expression) THEN (statement 1) ELSE (statement 2)
IF (expression) THEN (line number) ELSE (line number)
IF (expression) THEN (statement 1) ELSE (line number)
IF (expression) THEN (line number) ELSE (statement 2)

This form of conditional execution will cause statement one to be executed if the expression is true, but not statement two. However, if the expression is false then the second statement is executed, but not the first. IF THEN statements may be nested. If the statement is a line number then a GOTO (line number) is assumed.

Examples:

```
IF C=0 THEN LET C=1 ELSE LET C=0
IF WHY(Y%)=PI THEN PRINT PI ELSE PRINT WHY(W%)
```

++

Input/Output

PRINT (print expressions)

The PRINT statement is used to output numbers and strings to the standard output device or printer. Each expression is separated by either a comma or a semi-colon.

The comma will cause a TAB to the next multiple of column number 16. The semi-colon suppresses the auto-TAB and if it is the last separator will not cause a CR,LF to be output.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Examples:

```
PRINT                          (print a new line)
PRINT "Hello world"            (prints "Hello World"
                               on the terminal)
PRINT "Angle";angle            (print "Angle" and
                               the value of the real
                               variable 'angle' next
                               to it.

PRINT X%,TAB((Y'-1)*6);Z%
PRINT SPC(COL');NAME$
```

There are two special functions not available for other expressions. The TAB(<byte expression>)and the SPC(<byte expression>). TAB() outputs spaces until the column specified by the 'byte expression' is reached. If the current column is already past the specified column then no spaces will be output. The SPC() function outputs 'byte expression' spaces.

PRINT #<byte expression>,<print expressions>

This version of the PRINT statement causes output to be directed to the channel number specified by the value of the 'byte expression'. Channel Ø is the standard output device. Channel numbers exist between Ø and 12. See the section on OPEN NEW.

Examples:

```
PRINT #Ø,CHR$(12);              (do a form feed)
PRINT #1,TT%(X%,Y%)
PRINT #5,QUEST$(Q%,Y%+1)
```

INPUT [<literal string>],variable list

The INPUT statement requests data from the standard input device. The data entered must match the type of variables in the variable list. If the 'literal string' is found, it will be printed out to the standard output device. Variables may be subscripted (either single or double). Input to a variable stops at commas or illegal characters.

Examples:

```
INPUT "Enter the starting value",start'
INPUT "Enter the ending value  ",end'
INPUT A,B,C
INPUT "Enter your name ",nameofperson$(person%)
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

INPUT LINE [<literal string>],string variable

The INPUT LINE statement is just like the INPUT statement, however, all input goes to a single string variable (which can be subscripted). Everything entered will be assigned to the string including commas.

Examples:

```
INPUT LINE "Enter city, state, zip ",CSZ$
INPUT LINE BUFFER$
```

INPUT #<byte expression>,variable list
INPUT LINE #<byte expression>,string variable

These two forms of the INPUT and INPUT LINE statements allow reading data from other devices such as disk files. The 'literal string' is not allowed with this form. Input comes from the channel number specified in the 'byte expression'. See the section on OPEN, OPEN OLD, and CLOSE.

Examples:

```
INPUT #C',AA%(X%)
INPUT #1,ANSWER$(YNEXT%)
INPUT LINE #12,DATANUMBER
```

++

Loops

FOR <number assignment> TO <expression> [STEP <expression>]

Controlled loops can easily be handled by the FOR-NEXT structure. The 'number assignment' sets the starting value. The 'TO expression' sets up the ending value, and the optional 'step expression' sets the increment value which is automatically set to an increment of one if the STEP and step expression are not given.

FOR-NEXT loops may be nested up to 30 loops deep. Each of the expressions must be of the same type of number, ie... all floating, all byte, all word, all long, or all double.

Examples:

```
FOR X=1 TO 10
FOR X'=-9 TO 9 STEP 1
FOR X%=0 TO 9
FOR CONT#=100 TO 199
```

```
FOR YNEXT~=100 TO 0 STEP -1
```

NEXT (number variable)

The NEXT statement is used at the end of the loop. The 'number variable' is the same variable name used in the original 'number variable assignment' in the matching FOR statement. Integer loops are faster than floating point number loops. The 'variable' is called the index.

Examples:

```
NEXT X
NEXT X'
NEXT X%
NEXT CONT#
NEXT YNEXT~
```

++

Termination

END

The END statement will make the program exit. By default programs exit after the last statement is executed unless it is one of the transfer statements GOTO, RETURN, or CHAIN...etc.

Examples:

```
end
END
IF X>99 THEN END
```

STOP

The STOP statement causes a temporary halt. A message is issued to the standard output device. Entering a single character ("S") answer will make the program exit. Any other character will make the program continue. This statement can be used effectively with the TRACE, TRON, and TROFF directives to aid in program debugging.

Examples:

```
STOP
stop
IF COMMAND$="STOP" THEN STOP
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

External Programs

EXEC <string expression>

The EXEC statement calls the 'string expression' as a sub-program. When the called program is finished control is returned to the main program. In FLEX, the called program must not overlay the calling program, or else unpredictable results or side effects will occur.

Examples:

```
EXEC COMMAND$
EXEC "Ø.CAT.CMD"
EXEC "COPY /DØ/SINVENT /D1/INVENT/"+FNAME$
```

CHAIN <string expression>

The CHAIN statement is just like the EXEC statement except that when the called program completes control is NOT passed back to the calling program. In OS-9 the memory used by the calling program is returned to the system memory pool before the called program starts.

Examples:

```
CHAIN "MAINMENU"
CHAIN "LINKER.ROV"
CHAIN DO$
```

++

Sequential File I/O

OPEN OLD <string expression> AS <byte expression>
OPEN <string expression> AS <byte expression>

This statement opens the 'string expression' as a file name for read on the channel number specified by the 'byte expression'. The 'INPUT #' statement is used to read the data from the input channel opened. The channel numbers allowed for files are one to twelve (1 - 12).

Examples:

```
OPEN OLD DATA$ AS 1
OPEN OLD "MR"+FILE$+".DIC" AS 2
OPEN FILENAME$(X%) AS CH'
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Opening channel zero (Ø) is an exception allowed in FLEX for changing the standard output vector. Under FLEX the file opened is assumed to be a binary program which is loaded into memory. No checking is done to insure that it does in fact modify the standard output. Closing channel zero (Ø) will reset the output vector. The following example sends a form feed ($C) character to the printer:

```
1 OPEN "Ø.PRINT.SYS" AS Ø
2 PRINT #Ø,CHR$(12);
3 CLOSE Ø
```

Line 1 prepares the standard output to be redirected to the printer. Line 2 prints a form feed character. Finally line 3 closes channel zero which effectively resets the output vectors in FLEX. One last note, opening a device for printing on channel zero can be done with any of the OPEN statements.

OPEN NEW <string expression> AS <byte expression>

This statement opens the 'string expression' as a file name for write on the channel number specified by the 'byte expression'. The 'PRINT #' statement is used to write to the output channel opened.

Examples:

```
OPEN NEW DATA$ AS 2
OPEN NEW "DICTIONARY" AS 8
OPEN NEW MAINDIC$+".TXT" AS CHAN'
```

CLOSE <byte expression>[,<byte expression>,....]

The CLOSE statement shuts down a channel number which was previously opened by the OPEN NEW or OPEN OLD statements. Multiple channels may be closed by separating the 'byte expressions' with commas. Closing channel zero resets the I/O vectors.

Examples:

```
CLOSE CHAN'
CLOSE 1,2,3
CLOSE Ø,CHANNEL'(X%)
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

Associated File I/O

KILL (string expression)

The KILL statement deletes the disk file specified in the 'string expression'. This is a handy way to delete files.

Examples:

```
KILL OLDFILE$
KILL "DICKNAMES"+".TXT"
KILL "/D0/SYS/motd"
```

RENAME (string expression 1),(string expression 2)

The RENAME statement renames the file specified by 'string expression 1' to the file name specified in 'string expression 2'.

Examples:

```
RENAME OLD$,NEW$
RENAME FILE$+".TXT",FILE$+".BAK"
RENAME "LIST.CMD","L.CMD"
RENAME "DELETE.CMD","DEL.CMD"
```

++

Special Function Disk I/O

FLEX ONLY

SECTOR READ,(drive),(track),(sector),(byte array name)

The SECTOR READ statement reads a sector from the specified drive and puts the 256 bytes of data into the byte array. If the byte array has not been dimensioned it will be. The array may be accessed by any normal subscripts.

Examples:

```
SECTOR READ,DRIVE',0,3,C'(256)
SECTOR READ,DRIVE',TRACK',SECTOR',DATA'(256)
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

SECTOR WRITE,<drive>,<track>,<sector>,<byte array name>

The SECTOR WRITE statement writes the byte array out to the drive number specified. If the byte array does not exist then the array will be dimensioned and initialized to zeros, then it will be written out.

Examples:

```
SECTOR WRITE,0,0,3,C'(256)
SECTOR WRITE,D',T',S',BUFFER'(256)
```

With these two directives a programmer could write a program to read or write special disk formats. These two directives are supported only under FLEX.

++

Arrays

DIM <variable name(<word expression[,<word expression>]>),<ditto>

The DIM statement initializes the arrays and sets the array descriptors. The CAR (Close ARray) statement is used to recover the memory used by an array, at which point the array may be re-dimensioned.

Examples:

```
DIM A(10,10)
DIM QUESTIONS$(AMOUNT%)
DIM AM%(QQ%,CH%+1),AA%(QQ%+1)
```

CAR <variable name(0[,0])>

The CAR statement de-allocates the memory used by an array to prepare it for re-dimensioning. Arrays already dimensioned may not be re-dimensioned until they are closed by this statement.

Examples:

```
CAR A(0,0)
CAR QUESTION$(0)
CAR AM%(0,0)
CAR AA%(0)
```

The zero values for subscripts are used as dummies in order for the compiler to tell the size (one or two dimensions) of the array.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

AT $(hex address) (variable name)[((word expression)
[,(word expression)])]

The AT statement is a way of assigning variables to absolute addresses. If the variable name is an array it will be dimensioned at the address specified. Internally the address of the variable is kept for reference in expressions.

Examples:

```
AT $E090 TIMER%
AT $C080 INPUTBUF'(127)
AT $E040 PORT9'(3)
AT $B800 VIDEO'(51,25)
```

It can be used for I/O ports, graphics...etc.

++

Miscellaneous

POKE (word expression),(byte expression)

The POKE statement is used to put the 'byte expression' at the memory location specified by the 'word expression'. This is a dangerous statement so BE CAREFUL.

Examples:

```
POKE $E040,$15
POKE $CC00,RPT'
```

DPOKE (word expression 1),(word expression 2)

The DPOKE statement is used to put the value of 'word expression 2' at the memory location specified by 'word expression 1'. This is a dangerous statement so BE CAREFUL.

Examples:

```
DPOKE $CC2B,$B7FF
DPOKE $E004,$03FF
```

REMARK ....any text.....

The REMARK statement is included in K-BASIC to allow existing software to be compiled. Normally new programs will use the "*" comment lines for documenting purposes.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

REMARK may be shortened to REM.

Examples:

```
     REMARK --------------- ENTER NAME AND ADDRESS HERE
* Lines beginning with an asterisk are also a remark

     REM blank lines are also allowed
```

##

-- Intrinsic Functions --

++

Functions Returning a Real Number

The accuracy of the following functions is to 15 digits or very near 15 digits (within two or three digits). Level I of K-BASIC uses BCD math for real numbers so the more complex functions may run slower than expected. It was felt that if the math package prints 15 digits the functions should be as accurate.

EXP(real)

This function raises the natural log (2.71828182845904) to the power represented by the real expression (ie e^X).

Examples:

```
EXP(inf)
EXP(force-3)
```

The argument range is limited from -227.9 to +227.9 and the function generates values from -10^99 to 10^99. Any value under or over these limits will result in an error.

LOG(real)

This is the natural logarithm (base "e") of the 'real' number. If you want other bases such as base 10, use the formula:

Log of X to the base B = Log(X)/Log(B)

Examples:

```
LOG(30)
LOG(16)
```

SQR(real)

This is the square root function. The square root of a negative number is illegal.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Examples:

```
SQR(2)
SQR(100)
SQR(2)^2
```

COS(real)

This is the cosine function, which determines the cosine of angle of 'real', where 'real' is in radians.

Examples:

```
COS(PI)
COS(-PI)
COS(PI/2)
```

SIN(real)

This is the sine function, which calculates the sine of the angle, which is in radians.

Examples:

```
SIN(ANGLE(X%))
SIN(PI*2)
```

TAN(real)

This is the tangent function. It returns the tangent of the real expression, which is in radians.

Examples:

```
TAN((PI/180)*ANGLEINDEGRE)
TAN(A)
```

ATN(real)

This function returns the arctangent of the angle in radians. The value will be between -pi/2 and pi/2.

Examples:

```
ATN(X)
ATN(PI/4)
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

ABS(real)

The ABS() function returns a positive number whose value is the absolute value of the argument.

Examples:

```
ABS(-99)          returns 99
ABS(40-80)        returns 40
ABS(PI)           returns the value of PI
```

SGN(real)

The SGN() function returns a value representing the mathematical sign of the argument.

```
if real<0     Returns a -1
if real=0     Returns a  0
if real>0     Returns a  1
```

Examples:

```
SGN(-VALUE)   Returns -1 if VALUE is positive
SGN(-VALUE)   Returns 1 if VALUE is negative
SGN(-0)       Returns 0
SGN(4E99)     Returns 1
```

INT(real)

The INT() function returns a whole number whose value is not larger than the argument. In other words it 'chops off' the fractional part of the argument.

Examples:

```
INT(PI)           returns 3
INT(0.5)          returns 0
INT(-33.3)        returns -33
```

RND(real)

The RND() function returns a pseudo random number based upon the value given.

If the real argument <0

A new series of random numbers is started. For different negative values of the real argument, a different sequence is started, but if the argument is the same the function will keep starting the same random sequence. In other words the negative number is used as the seed for a new series of

random numbers.

If the real argument =Ø

Returns a new random number each time called. This is the normal value of the argument for the RND() function.

If the real argument)Ø

Returns the last random number generated.

Examples:

```
RND(-.Ø2933)        start a new series
RND(Ø)              returns a new random number
RND(1)              get the same random number
```

VAL(N$)

The VAL() function returns a real number from converting the string N$ to a real number.

Examples:

```
VAL(Number$)
VAL("Ø.315")
VAL(DSTR$(D~))      convert a double to real
```

PI

The value of 3.14159265358979 is returned. The value PI is the angle in radians for the angle 18Ø degrees. Therefore PI/2 is the same as 9Ø degrees and PI*2 is the same as 36Ø or Ø degrees.

++

Functions Returning a String

STR$(real)

The STR$() function is just the opposite of the VAL function. It converts the real number argument to a string.

Examples:

```
STR$(PI)
STR$(WVAL("$EØ4Ø"))
```

BSTR$(byte)

The BSTR$() function is just the opposite of the BVAL() function. It converts the value of a byte expression to a string.

Examples:

```
BSTR$(12)
BSTR$(AR'(X%))
```

WSTR$(word)

The WSTR$() function is just the opposite of the WVAL() function. It converts the value of a word expression to a string.

Examples:

```
WSTR$($E000)
WSTR$(INDEX%)
```

LSTR$(long)

The LSTR$() function is just the opposite of the LVAL() function. It converts the value of the long expression to a string.

Examples:

```
LSTR$(LONGWORD#)
LSTR$(4000+Y#)
```

DSTR$(double)

The DSTR$() function is just the opposite of the DVAL() function. It converts the value of the double expression to a string.

Examples:

```
DSTR$(0)
DSTR$(DUMPADD~)
```

LEFT$(string,byte)

The LEFT$() function returns a substring of the string expression starting at the beginning of the string to the character position specified by the byte expression. If the byte expression is larger than the length of the string then the whole string is returned.

Examples:

```
LEFT$(QUESTION$,1)
LEFT$(STR$(NUMBER),LEN(STR$(NUMBER))-1)
```

RIGHT$(string,byte)

The RIGHT$ function returns a substring of the string expression starting at the location specified by the byte expression from the right end of the string expression to the end of the string.

Examples:

```
YEAR$=RIGHT$(DATE$,2)
ZIP$=RIGHT$(CSZ$,5)
```

MID$(string,byte1,byte2)

The MID$() function returns a substring of the string expression starting at the character position specified by the byte expression 1 and for the number of characters defined by byte expression 2. Out-of-range errors result in defaults of null strings or the remaining portion of the string.

Examples:

```
MID$(DATE$,4,3)
MID$(STR$(PI),2,LEN(STR$(PI))-3)
```

CHR$(byte)

This function converts the value of the byte expression to a string of one character whose value is the same as the byte expression.

Examples:

```
CHR$(27)          escape
CHR$(12)          form feed
```

INCH$(byte)

The INCH$() function gets a single character from the channel specified by the byte expression. A string with a length of one character is returned.

Examples:

INCH$(0)	get a character from the standard input.
INCH$(CH')	read from a file.

ROM$(word)

This function converts the value of the word expression to a ROMAN number. If the number is positive, then upper case roman numerals will be used (MDCLXVI), however, if it is negative, lower case numerals will be used (mdclxvi). This is a handy way to express dates, etc. The argument is limited to an absolute maximum value of 3999.

Examples:

ROM$(1500)	returns MD
ROM$(-1984)	returns mcmlxxxiv

DATE$

This string is not a function, but instead returns the system date as a string in the form "DD-MON-YY".

Examples:

DATE$	returns 04-JUL-84 if the date is the 4th of July, 1984

PARAMETER$

This string is not a function. It contains the parameters given on the command line when the program was called. It returns the complete parameter portion up to but not including the (cr) or the program name.

Example:

```
FLEX: STAMP PIERSON
OS9:  stamp PIERSON

PARAMETER$ is equal to "PIERSON"
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

++

Functions Returning a Byte Number

BVAL(string)

This function converts the string expression to a byte number. If illegal characters are encountered the processing stops and the remaining value is returned.

Examples:

```
BVAL("$FF")          the value of -1
BVAL(WSTR$(ERR))     convert the error number
                     to byte size
```

ASC(string)

The ASC() function returns the byte integer whose value is the ASCII value of the first character of the string expression. If the string is a null then the returned value is a zero.

Examples:

```
ASC(INCH$(0))
ASC(Q$(X%))
```

LEN(string)

The LEN() function returns a byte integer whose value is the length in characters of the string expression.

Examples:

```
LEN("HELLO")         returns 5
LEN("")              returns 0
LEN(INCH$(0))        always returns 1
```

POS(byte)

The POS() function returns the byte value of the current position of the channel number specified by the byte argument. Channel numbers range from 0 to 12.

Examples:

```
POS(0)               returns the current position
                     of the standard output
POS(CH')
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

PEEK(word)

The PEEK() function returns the byte value of the memory location specified by the argument.

Examples:

```
PEEK($CC00)
PEEK(PORT%)
PEEK(PORT%+1)
```

INSTR(byte,string1,string2)

The INSTR() function does a substring search of string2 on the string1 starting at the character position specified by the byte expression. The value returned is the character position of the match. A zero is returned if no match is made.

Examples:

```
INSTR(0,"HELLO WORLD"," ")     returns 6
INSTR(CL',COMMAND$,KEYWORD$)   look for a command
```

CHANNEL

This predefined variable contains the last channel number used in any I/O statement. It can be used with error checking. CHANNEL is a byte expression type.

Examples:

```
PRINT "Error in CHANNEL :";CHANNEL
```

++

Functions Returning a Word Number

WVAL(string)

The WVAL() function converts the string expression to a word size integer.

Examples:

```
WVAL(DSTR$(ABSD~))  convert a double to word
WVAL(STR$(NUM))     convert a real to a word
```

Copyright (c) 1984, LLOYD I/O All Rights Reserved

HEX(string)

The HEX() function converts the string expression to a word integer. The string must consist of the hexadecimal characters 0-9 and/or A-F. It must be at least one digit and up to four characters.

Examples:

```
HEX("E090")
HEX(BSS$)
```

DPEEK(word)

The DPEEK() function returns the word value of the memory locations specified by the argument. The argument is the address of the first byte, the most significant byte of the word returned.

Examples:

```
DPEEK($CC2B)
DPEEK(ADDRESS%)
```

WRND(word)

The WRND() function returns a pseudo random number between zero and the value of the argument. This is a useful function for games. The range of arguments is limited between 1 and 32767.

Examples:

```
WRND(255)
WRND(8000)
WRND(100)
WRND(MAXGAME%)
WRND(0)            return the last random number
WRND(-1021)        seed the random number
```

A negative argument is used as a seed to the random number generator. It should be one of the prime numbers. After the seeding, a random number is returned. An argument with a zero value will return the last random number generated.

FRE(0)

The free memory function returns the amount of memory available for arrays, strings, and file control blocks.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Example:

```
FRE(0)                this is the only form
```

If FRE() returns a negative value, it means that there are more than 32,767 bytes free. It really means there are 32767+(FRE(0)+32768) bytes free.

PTR(variable name)

The PTR() function returns the word integer whose value is the address of the variable name. Array references are not allowed.

Examples:

```
PTR(PORTB')
PTR(X$)
```

ERR

This is not a function, but is a word variable whose value is that of the last error encountered in a program. It can be used in conjunction with error trapping.

Examples:

```
IF ERR<>8 GOTO BADDATA
IF ERR=8 THEN CLOSE 1
```

ERL

The variable ERL contains the source code line number which is automatically updated at run time if the trace mode is turned on (TRON).

Example:

```
PRINT "ERROR #";ERR;" IN LINE #";ERL
```

++

Functions Returning a Long Number

LVAL(string)

The LVAL() function converts the string expression to a long integer. Using it in combination with the assorted STR$() functions allows conversion from different sized

integers and real numbers.

Examples:

```
LVAL(STR$(REALNUMBER))
LVAL(WSTR$($CC2B))
```

++

Functions Returning a Double Number

DVAL(string)

The DVAL() function converts the string expression to a double integer value. Again, using it in conjunction with the STR$() functions allows number size conversions.

Examples:

```
DN~=40+DVAL(STR$(ANGLE/180*PI))
PRINT DVAL(Vahs$)
```

##

-- Compiler Errors --

The following is a list of the error messages. Errors are reported in a special way. The line in which the error ocurred is printed. Then a blank line is printed with the character "^" in the position of the error. The "^" is said to point just past where the error was found. The error message is displayed. If a line has several errors they will be reported several times just as described here.

Example:

```
00072 LOOP          PRINT A(X)
                          ^
*** ILLEGAL SUBSCRIPTING
```

ILLEGAL CHARACTER(S)

Characters found that are not allowed or are undefined for use. Make sure literal (constant) strings begin and end with the double quote (").

SYNTAX ERROR

A general purpose error message used almost everywhere. Look at the expected syntax for directives, statements, or functions.

INPUT STATEMENT WITHOUT VARIABLE

INPUT and READ statements always have to have an assignment variable list.

ASSIGNMENT ERROR

Probably a missing equals sign ("=").

BAD VARIABLE TYPE

The variable type used in the expression is not legal.

EXPRESSION TOO COMPLEX

The expression has too many parentheses going too deep.

ILLEGAL OPERATOR

An operator used with a constant or variable type

that is illegal.

ILLEGAL CONSTANT TYPE

A constant real is being used with an integer expression or a with character string expression or some other type of mismatch with constants.

ILLEGAL SUBSCRIPTING

Only word integer expressions are allowed as subscripts in DIMensioned variables.

FUNCTION CALL ERROR

The expected expression is of the wrong type or is missing.

OPERAND MISMATCH IN EXPRESSION

The wrong type of variables or constants is being used. Only by the use of functions can different types be mixed. An assignment mismatch could also be in error.

FLOATING POINT UNDER/OVERFLOW

The constant real number is over or under range.

IF THEN NESTING STACK OVERFLOW

The IF THEN conditional statement is nested too deep.

FOR NEXT LOOP NESTING STACK OVERFLOW

The nesting limit to FOR NEXT statements is 30 deep.

"FOR" STATEMENT COUNTER ERROR

The internal FOR statement counter has been fouled up. This error should not appear.

"NEXT" STATEMENT WITHOUT MATCHING "FOR"

FOR - NEXT statements must go together to form a loop.

COMMA "," NOT FOUND

The expected comma is missing.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

COMMA EXPECTED

A missing comma has been detected.

UNBALANCED PARENTHESES IN EXPRESSION

An opening or most likely a closing parenthesis "()" is missing or has been mis-placed.

RIGHT ')' NOT FOUND

The closing parenthesis is missing.

LEFT '(' NOT FOUND

The opening parenthesis is missing.

OPEN STATEMENT SYNTAX ERROR

The expected string or byte expressions are in syntax error.

CLOSE STATEMENT ERROR

The byte expressions have a syntax error.

MEMORY OVERFLOW

All available work space memory has been used. Try assigning more memory (OS9), or, try dividing up the program into smaller chunks. Sometimes this error occurs when the expression is too complex.

MULTIPLY DEFINED LABEL OR LINE NUMBER

A label has been used twice. Each label must be unique.

UNDEFINED LABEL OR LINE NUMBER

A label has been referenced that has never been defined. Labels always are defined by starting them in column 1 of the line in which they are to be defined.

LINE NUMBER OR SYMBOL ERROR

A line number was expected as the label type.

SYNTAX ERROR IN STRAIGHT COPY

An error in the file KSWORDS has caused a problem. If you have this error please give us a call or write because this error should never occur.

VARIABLE NOT DEFINED IN PASS 1

An undefined variable was found. This error should not occur. Again call us if it does.

LIBRARY FILE CALLING ERROR

A problem with the file name used with the LIB directive has ocurred.

##

-- Assembly Errors --

The supplied assembler supports 22 error messages which are always printed BEFORE the offending line. They are as follows:

1. MULTIPLY DEFINED SYMBOL
2. UNDEFINED SYMBOL
3. ILLEGAL CHARACTER(S)
4. UNRECOGNIZABLE MNEMONIC OR MACRO
5. MNEMONIC REQUIRES A LABEL
6. RELATIVE BRANCH TOO LONG
7. ILLEGAL ADDRESSING MODE
8. USE (PATH) OR LIB (PATH) NESTING LEVEL TOO DEEP
9. OUT OF MEMORY WHILE DEFINING A MACRO
10. OUT OF MEMORY WHILE CALLING A MACRO
11. ILLEGAL REGISTER SPECIFIED
12. ABORTING..MEMORY OVERFLOW
13. MNEMONIC CANNOT HAVE A LABEL
14. PARAMETER SUBSTITUTION ERROR
15. ILLEGAL NESTING
16. ILLEGAL EXPRESSION
17. ILLEGAL 'CPM' CALL
18. MULTIPLY DEFINED MACRO
19. UNBALANCED PARENTHESES IN EXPRESSION
20. ILLEGAL LABEL OFFSET SPECIFIED
21. ILLEGAL SYMBOL CHARACTERS
22. PHASING ERROR DETECTED

Error messages are printed in the following manner:

*** error message IN LINE # xxxx OF "file name.ext"

The line number is the actual line number of the line which was read from the current file, even if it is a nested library file. This feature tells you exactly which line and which file caused the error.

NOTE: if you should get any type of assembly errors please send us a listing of the BASIC program that is being compiled on an OS9 or FLEX diskette in SINGLE/SIDED SINGLE/DENSITY format if possible. We will try to correct the problem and send you an update, however, we make no gauantee to do so. It is assumed that you will know how to take action to correct these errors.

##

-- Run-Time Errors --

The run time errors occur when a program that has been compiled is executed. This means that the compiler itself and the source code software are not available in memory to aid in the reporting of errors. A simple error reporting scheme has thus been invented. It reports only the error number:

BASIC ERROR #73

If a programmer feels ambitious he could replace it with a more complex reporter with text error messages and the like.

In FLEX the error numbers 0 to 29 are the FMS error numbers. When an error of 29 or less is reported the FLEX error reporting system is also called. I.E. the "ERRORS.SYS" system is used.

In OS9 error numbers from 200 to 255 are the system error numbers. The PRINTERR reporter will be used if implemented.

Therefore, in reality, the error numbers used by the K-BASIC run-time package are from 30 to 199 and from 256 to 65535. The following is a description of each error number.

30 - DATA TYPE MISMATCH

Wrong type of DATA found in a READ statement.

31 - OUT OF DATA IN "READ"

The last DATA has already been read by a READ statement. The RESTORE statement can be used to reset the DATA pointer to the beginning of the DATA statements.

32 - BAD ARGUMENT IN "ON" statement

The expression is out of range. Either a zero result or a value too large for the number of labels.

40 - BAD FILE OR CHANNEL NUMBER USED

The byte expression used for channel numbers is out-of-range. Channel numbers 1 to 12 are used for files, channel 0 is used for standard input and output redirection.

52 - SYNTAX ERROR

The general purpose syntax error.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

62 - "FOR-NEXT" NEST ERROR

Usually the result of assembling a program that was compiled with an error.

73 - ILLEGAL EXPRESSION

Each expression is checked to see if the first byte is of the correct type for the expression about to be calculated. Most likely the X register is pointing somewhere else other than the expected location in memory.

77 - ARRAY REFERENCE OUT OF RANGE

The subscript value is greater than the DIMensioned size. Array references range from zero to the value in the dimensioned size. Arrays are dimensioned at run-time, not compile time.

80 - MEMORY OVERFLOW

Not enough memory available for calculating an expression or opening a file.

81 - ARRAY OVERFLOW

Not enough memory available for dimensioning an array.

83 - STRING TOO LONG

Not enough memory available for saving or assigning a string. String arrays are an array of pointers (two bytes each) which point to a string. If the address is zero, then no string has been assigned.

100 - EXPRESSION TOO COMPLEX
101 - OVERFLOW OR UNDERFLOW IN MATHEMATICAL OPERATION
102 - ARGUMENT TOO LARGE
103 - DIVISION BY ZERO
105 - NEGATIVE OR ZERO ARGUMENT FOR "LOG"
107 - IMAGINARY SQUARE ROOT
108 - CONVERSION ERROR (NUMBER TOO LARGE)

The above errors result in expression operands that are illegal in value.

300 - PROGRAMMABLE BREAK (CONTROL-C) TRAP
OR THE "ESCAPE RETURN" SEQUENCE TRAP OF FLEX

A control-C will cause the program to abort. Also a "ESCAPE RETURN" sequence will cause an abort.

The ON ERROR GOTO (label) statement can be used to redirect error handling to trap out errors from any of the above errors.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

##

-- Program Debugging --

K-BASIC has limited facilities for debugging a program. Some of these should be used in the normal technique of error trapping or error handling. These are the statements available for this use:

ON ERROR GOTO <label>	sets the error trap location
RESUME <label>	returns from an error trap
STOP	temporary program stop

The following predefined variables can be used in an error handling routine:

ERR	the last error number
ERL	the source code line number
CHANNEL	the last I/O channel used

The following statements are used for program debugging:

TRON	turns on the trace mode
TROFF	turns the trace mode off
TRACE	prints the current trace values

It is good programming practice to include in your program methods of recovering from errors. These include I/O, data conversion, and expression errors.

I/O (input/output) errors are the type where the end of a file is found, or all disk space has been used, or the file name could not be found, or already exists... etc.

Data conversion errors are the type where illegal characters are found when converting a string to a number. The only allowed characters that can terminate this type of conversion are a <CR> or a comma (,).

Expression errors come from illegal arguments for functions, divide by zero...etc. One type of error is the illegal expression which is usually a problem internal to the K-BASIC run-time package.

All error traps MUST exit with the RESUME <label> statement. The label must be in the main flow of the program. This means it cannot RESUME to a subroutine. RESUME fixes the program stack pointer to the top of the stack and all return addresses are lost. The ON ERROR GOTO <label> can be set as many times as needed in a program. If you want to reset it back to the normal error reporting system then use:

ON ERROR GOTO

The form:

ON ERROR GOTO Ø

will not work.

Your error trap can check the values of ERR, ERL and CHANNEL. These predefined variables provide information that enables the program to take the correct action for error processing.

ERR is a word variable that contains the error number of the last error. It can be checked for any error, such as I/O, data conversion, or expression errors.

ERL is a special predefined word type variable. If the trace mode was used it will contain the source code line number of the error. It does not contain a (label) or (line number), but the physical line number encountered during the compile. If you are using an editor that has line numbers, these would be the values in ERL. The line numbers at the left edge of the compiler's listing are the line numbers used. There will be a little more on this later. For now though it can be used to tell which line caused the error.
For example:

```
PRINT "ERROR ";ERR;" IN LINE ";ERL
```

will print the error number and the source code line number that caused the error.

CHANNEL is a predefined byte type variable. It contains the channel number of the last I/O function. Normally it will contain a zero value for standard input and output statements like INPUT and PRINT. But if another I/O channel was used it will contain that channel number. It is limited in range between zero and twelve (Ø-12) as legal channel numbers. Any other value will be illegal. When doing I/O error processing it is a good idea to check CHANNEL to see on which channel the error ocurred.

```
Example:  PRINT "ERROR ";ERR;" ON CHANNEL ";CHANNEL
```

Now on to program debugging. The TRON and TRACE statements turn on the trace mode. This forces the compiler to imbed the source code line numbers into the run-time program. As the program is executing the predefined variable ERL is constantly being updated with the source code line number of the statement about to be executed. Therefore, ERL will NOT contain the correct source code line number in an error trap routine. To avoid this problem the error trap routine must use TROFF before the label associated with the start of the error trap. If this is not done ERL will be updated with the source code line numbers of the error trap itself. This would inhibit tracing an error to a specific source code line.

Copyright (c) 1984, LLOYD I/O All Rights Reserved

Example:

```
          TROFF
EHERE     IF ERR<>8 GOTO NOERRS
          IF CHANNEL<>1 GOTO NOERRS
          CLOSE 1
          RESUME MAIN

NOERRS    PRINT "ERROR ";ERR;"IN LINE";ERL;"ON CHANNEL";CHANNEL
          STOP
          RESUME MAIN
```

The TRACE statement's main purpose is to print the actual memory address of the next statement and its source code line number. It also turns on the trace mode. The statement "TRACE" can be placed at key points in a program for debugging purposes.

Normally a programmer will place PRINT statements at various locations in a program to display values for debugging purposes. This is the best way to debug a program without getting a screen full of meaningless source code line numbers.

Here is a very clever way to process multiple channel errors:

Example:

```
          TROFF
ETRAP     IF ERR>29 GOTO NONIOERR
          IF CHANNEL=0 GOTO ERRIO0
          ON WVAL(BSTR$(CHANNEL)) GOTO EC1,EC2,EC3,EC4,\
EC5,EC6,EC7,EC8,EC9,EC10,EC11,EC12

EC1       PRINT #1,"EOF"
          CLOSE 1
          RESUME MAIN
EC2       REM
EC3       REM
EC4       REM

* continue with all channel labels

EC12      REM
          PRINT "CHANNEL ";CHANNEL;" IS NOT BEING USED"
          STOP
          RESUME MAIN
NONIOERR  PRINT "ERROR ";ERR;"IN LINE ";ERL
          STOP
          RESUME MAIN
ERRIO0    PRINT "ERROR ";ERR;" IN LINE";ERL;" IN CHANNEL 0"
          STOP
          RESUME MAIN
```

Debugging K-BASIC programs will take more time than debugging programs used with an interpreter. Because of this you will have to take a little more care in planning and coding programs to be

compiled. Since K-BASIC syntax is so close to existing interpreters you may want to use them first, then modify your programs for K-BASIC. At minimum, a typical compile (including the assembly) will take about three to four minutes, more if your program is very long. This could slow you down quite a bit if you make many mistakes.

If you have any comments please write, or call LLOYD I/O at:

(503) 666-1097

LLOYD I/O
19535 NE GLISAN STREET
PORTLAND, OR 97230 USA

##

-- Output Files --

The compiler outputs all information in assembly language source code. There are several files into which the different types information are written:

".ASM"

This is the main file to be assembled. It is a short file which has LIBrary file calls to all of the rest of the source code files. It defines the originate of the storage and the program. It also calls the macro "MAIN" which defines the startup sequence of instructions.

".EQU"

This file defines the labels that are used in conditional assembly throughout all the source code files. The labels are defined in the compiler's special input file called "LABELS".

".RMB"

This file defines all variables used in a BASIC program. All string and array references are defined as a control block which contains at minimum an address of where the data for the variable is stored.

".SUB"

This is the main file used to write the assembly source code translations for each program line in the BASIC program. It mostly consists of macro calls with parameter passing.

".CON"

This file defines all constants and tokenized expressions except constants from DATA statements. Constants are written out in their natural form.

".COD"

This file contains the data from DATA statements. It is written out one piece at a time separated by commas to form one very large string of data. The last character is a CR ($0D).

Copyright (c) 1984, LLOYD I/O All Rights Reserved

I N D E X

ABS() 48
Arrays 43
ASC() 53
ASSEMBLE 30
Assembly Code Passing 29
Assembly Errors 62
Assignment 19
Assignment Statements 32
Associated File I/O 42
AT 44
ATN() 47
BSTR$() 50
BVAL() 53
Byte Functions 53
CAR 43
CHAIN 40
CHANNEL 54
CHR$() 51
CLOSE 41
CODE 29
Compiler Errors 58
Conditional Execution 35
Constants 16
Conventions 13
COS() 47
DATA 32
DATE$ 52
Debugging 65
Debugging Aids 28
Definitions 10
DIGITS 27
DIM 43
Dimensioning 18
DIR 30
DIRectory Defaults 30
Double Functions 57
DPEEK() 55
DPOKE 44
DSTR$() 50
DVAL() 57
END 39
ENDCODE 29
ERL 56
ERR 56
EXEC 40
EXP() 46
Expressions 15
External Programs 40
FLEX System Requirements 4
FOR TO STEP 38

FRE(Ø) 55
Fundamentals 13
Getting Started 5
GOSUB 33
GOTO 33
HEX() 55
History 3
IF GOTO 35
IF THEN 35
IF THEN ELSE 36
INCH$() 51
INPUT 37
INPUT # 38
INPUT LINE 38
INPUT LINE # 38
Input/Output 36
INSTR() 54
INT() 48
Invoking 6
Invoking the Assembler 8
KILL 42
LEFT$() 50
LEN() 53
LET 32
LIB 30
LIBrary Files 30
LINE 26
LOG() 46
LOG() 46
Logical Operators 21
Long Functions 56
Loops 38
LSTR$() 50
LVAL() 56
Math Operators 21
MEMORY 25
MID$() 51
Modifing Memory 44
NAME 25
NEXT 39
ON ERROR GOTO 34
ON GOSUB 34
ON GOTO 34
OPEN NEW 41
OPEN OLD 40
Operators 21
OS9 System Requirements 4
Output Files 69
PAGE 28
Pagination 28
PARAMETER$ 52
PEEK() 54
PI 49
POKE 44
POS() 53

Precedence 23
PRINT 36
PRINT # 37
Program Debugging 65
Program Location 25
PTR() 56
READ 33
Real Functions 46
Relational Operators 22
REMARK 44
Remarks 23
RENAME 42
RESTORE 33
RESUME 35
RETURN 34
RIGHT$() 51
RND() 48
ROM$() 52
Run-Time Errors 63
Sample Programs 4
SECTOR READ 42
SECTOR WRITE 43
Sequential File I/O 40
SGN() 48
SIN() 47
Special Disk I/O 42
Special Length Control 26
START 26
STOP 39
STORAGE 26
STR$() 49
STRING 27
String Functions 49
String Operators 22
STTL 28
System Requirements 3
TAN() 47
Termination 39
TRACE 29
Transfer and Control 33
TROFF 29
TRON 28
TTL 28
User Prerequisites 5
VAL() 49
Variables 17
Word Functions 54
WRND() 55
WSTR$() 50
WVAL() 54

Copyright (c) 1984, LLOYD I/O All Rights Reserved