BASIC09/68000
PROGRAMMING LANGUAGE
REFERENCE MANUAL

Revision B
Publication date: July 1985
Publication Editor: Walden Miller

Microware System Corporation
1866 N.W. 114th Street
Des Moines, Iowa 50322 U.S.A.
(515)224-1929

PMN-BAS68

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL

INTRODUCTION

BASIC09 is an enhanced structured BASIC language programming system. It was specially created for the 68000 Microprocessor.

In addition to the standard BASIC language statements and functions, BASIC09 includes many of the most useful elements of the PASCAL programming language. This allows programs to be modular, well-structured and use sophisticated data structures. It permits full access to almost all of the OS-9 Operating System commands and functions so it can be used as a systems programming language.

BASIC09 is unusual in that it is an **Interactive Compiler**. It has the fast execution speed typical of compiler languages and the ease of use and memory space efficiency typical of interpreter languages.

BASIC09 includes a powerful text editor, a multi-pass compiler, a run-time interpreter, a high-level interactive debugger, and a system executive. Each of these components was carefully integrated so the user "sees" a friendly, highly interactive programming resource. It provides all the tools and helpful facilities needed for fast, accurate creation and testing of structured programs.

These features make BASIC09 an ideal language for many applications: scientific, business, industrial control and education.

BASIC09 FEATURES

Structured, recursive BASIC with PASCAL-type enhancements:

- Allows multiple, independent, named procedures
- Procedures called by name with parameters
- Multi-character, upper or lower case identifiers
- Variables and line numbers local to procedures.
- Line numbers optional
- Automatic linkage to ROM or RAM "library" procedures
- PACK command compacts program and provides security
- PRINT USING with FORTRAN-like format specifications

Extended Control Structures (with Unique Closure Elements):

- IF...THEN...[ELSE...] ENDIF
- FOR...TO...[STEP]...NEXT
- REPEAT...UNTIL...
- WHILE...DO...ENDWHILE
- LOOP...ENDLOOP
- EXITIF...THEN...ENDEXIT

High-speed, high-accuracy math:

- 14-decimal-digit 64-Bit-binary floating point
- Full set of transcendentals (SIN, ASN, ACS, LOG, etc.)

Extended data structures:

- 5 basic data types: BYTE, INTEGER, REAL, BOOLEAN and STRING.
- One, two, or three-dimensional arrays
- User-defined complex structures and data types

Powerful interactive debugging and editing features:

- Integral full-feature text editor
- Syntax error check upon line entry and procedure compile
- Trace mode reproduces original source statements
- Renumber command for line numbered procedures

THE HISTORY OF BASIC09

BASIC09 was conceived in 1978 as a high-performance programming language to demonstrate the capabilities of the 6809 microprocessor to efficiently run high-level languages. BASIC09 was developed at the same time as the 6809 under the auspices of the architects of the 6809. The project covered almost two years, and incorporated the results of research in such areas as interactive compilation, fast floating point arithmetic algorithms, storage management, high-level symbolic debugging and structured language design. These innovations give BASIC09 its speed, power and unique flavor.

BASIC09 was commissioned by Motorola, Inc., Austin, Texas, and developed by Microware Systems Corporation, Des Moines, Iowa. Principal designers of BASIC09 were Larry Crane, Robert Doggett, Ken Kaplan, and Terry Ritter. The first release was in February, 1980.

Excellent feedback, thoughtful suggestions, and carefully documented bug reports from BASIC09 users all over the world have been invaluable to the designers in their efforts to achieve the degree or sophistication and reliability BASIC09 has today.

INTRODUCTION

CONCERNING THIS MANUAL

This manual is divided into two parts: the BASIC09 Tutorial and the BASIC09 Reference Manual.

The tutorial section is written for beginning programmers having little experience with PASCAL or other high level languages. Beginning programmers should work through the examples given to help familiarize themselves with BASIC09 control structure.

Readers having adequate programming skills are urged to browse the tutorial for a feeling of the BASIC09 environment. A complete index is provided for easy use of the reference section.

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL

TABLE OF CONTENTS

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL

TABLE OF CONTENTS

PART 2: BASIC09 REFERENCE GUIDE

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

PART I - THE BASIC09 TUTORIAL

- Basic Program Construction
- Complex Data Types and Subroutines
- Program Optimization

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL
CHAPTER 1
BASIC PROGRAM CONSTRUCTION

WHAT IS A PROGRAM?

A computer program is a series of directions to manipulate information. You can write a program to list, copy, sort or merge information. Programs can perform arithmatic functions. The only limit to the ability of what a program can accomplish, is the type of directions available and the amount of skill used in putting them together in a program.

To fully utilize a programming language (such as BASIC09) a programmer must understand the directions available. This includes knowing what directions are available and what these directions can actually accomplish.

When writing a program, it is important to understand what you want it to accomplish and what type of information will be used by the program. Many programs do not initially work as intended. Others work correctly if given the correct type of data, but fail when used with incorrect types.

There is a process that all programs are subject to:

STEP 1. Writing the original program.

STEP 2. Running the original program.

STEP 3. If the program does not run or produce the correct results, examine it and change whatever seems to be the problem. This is called debugging a program. If the program does run and produces the correct results, you are finished with this individual program.

STEP 4. Run the program again. Go to STEP 3.

As you can see, programming can be frustrating during the testing and debugging process. However, by understanding the use of the language's directions and features, debugging time can be cut to a minimum.

GETTING STARTED

When you turn on your BASIC09 computer, it will print some heading information and then display the OS9 prompt: "$". To enter the BASIC09 environment, type "basic09" after the "$" prompt and hit the <return> key. The system will respond by printing the following:

```
$ basic09
Copyright 1984 by Microware
Reproduced Under License
BASIC09/68000
READY
B:
```

The "B:" prompt indicates that you are in the "system" mode of BASIC09. While in system mode, you may create, list or kill procedures (the BASIC09 name for programs). For now, we are interested in creating a procedure.

To create a new procedure, you must command the system to enter edit mode. To do this, type "e", followed by the name of the procedure you wish to create after the "B:" prompt. If no procedure name is specified after the "e" command, the system will assign the name or "PROGRAM" to the new procedure.

BASIC09 will respond by printing "PROCEDURE" followed by the name you have specified (or "PROGRAM" if no name was specified). On the next line, it will print an "*" signifying the current edit line in the procedure. On the next line, the edit mode prompt, "E:", will be displayed:

```
B: e myprogram
PROCEDURE myprogram
*
E:
```

Procedure names may contain from 1 to 28 upper or lower case letters, numbers or special characters (as listed below). While the file name may begin with any of the following characters or digits, the file name must contain at least one number or letter. Within these limits, a name may contain any combination of the following:

```
-upper case letters:  A-Z
-lower case letters:  a-z
-numbers:             0-9
-underscores:          _
-dollar signs:         $
-periods:              .
```

In edit mode, the first character of each line is reserved for edit commands. If you forget to type an edit command, BASIC09 will respond with "What?".

The most important edit command is the <space>. Placing a <space> in the first character position, saves the line of the procedure that immediately follows.

THE INPUT AND PRINT STATEMENTS

The two most basic statements in BASIC09 are **INPUT** and **PRINT**. These allow a procedure to receive and print data respectively.

INPUT accepts data during the execution of a procedure. The data is normally read from your terminal (the standard input device). PRINT outputs the text or the values given on the print line. The output is normally sent to the terminal (the standard output device).

Both the input and output can be redirected, so that input can be read from a file, or output can be sent to a printer, for example.

To use the PRINT statement in our procedure "myprogram", use the <space> command and enter the following line:

```
E: PRINT "type your name"
```

When BASIC09 executes "myprogram", this line will tell it to print on the terminal screen all of the characters between the quotes.

To use the INPUT statement in "myprogram", use the <space> command and enter the following line:

```
E: INPUT name$
```

This line, when executed, will command BASIC09 to wait for a line of text to be received from the keyboard. BASIC09 will accumulate text from the keyboard, character by character, until a <return> ends the line. This text will be saved in the memory reserved for the variable "name$".

To finish this program enter the final two lines as follows:

```
E: PRINT "Hi, ";name$;". It's been nice talking to you."
*
E: END
```

The third line of "myprogram" starts out similarly to the first line: <space> PRINT "something in quotes". However, the semicolon following the end quote after "Hi, " tells BASIC09 that something else is to be printed on this line. BASIC09 inserts the text it saved from the INPUT statement for the variable "name$". Again the next semicolon informs BASIC09 that there is more to be printed.

When a PRINT statement contains multiple values (as in the third line of "myprogram"), it will print them out consecutively. Each of these values must be separated by a comma or a semicolon. If the separator is a comma, BASIC09 will move to the next 16-column "tab stop" before printing the next value. If the separator is a semicolon, no space will separate the fields.

The last line of "myprogram" ("END") tells BASIC09 stop executing the procedure and return you to the system mode. The END statement is optional. BASIC09 will return to system mode upon executing the last line of code within the procedure.

To review your now complete procedure, type "l*". This is the edit mode command to list all the lines of the procedure.

```
E:l*
PROCEDURE myprogram
 0000      PRINT "type your name"
 0012      INPUT name$
 0017      PRINT "Hi, "; name$; ". It's been nice talking to you."
 0035      END
*
E:
```

NOTE: The editor has added some information which you did not type in. The numbers to the left of the program are "I-code addresses". These are the actual memory locations where each line begins relative to the start of the procedure. These numbers may be strange because they are in hexadecimal (base 16). The I-code addresses are important because when BASIC09 finds an error in a procedure, it will convey as much information as it has concerning the error. One such piece of information is the I-code address. I-code addresses are automatically supplied by BASIC09.

All that is now left to do, is to run the procedure. To run a procedure you must be in system mode. Type "q". This "quits" the editor and returns you to system mode:

```
E:q
READY
B:
```

Now type "run myprogram" or "run". The **RUN** command (without a procedure name) instructs BASIC09 to execute the last edited procedure (in this case "myprogram"). In this instance both commands cause the same procedure to be run:

```
B: RUN myprogram
type your name
? walden
Hi, walden. It's been nice talking to you.
READY
B:
```

NOTE: The question mark ("?") is the normal input prompt to tell the user that the program is waiting for input.

SAVING PROCEDURES

Now that you have written your first program, you will want to save it. Currently, "myprogram" exists only in the BASIC09 workspace.

The workspace is the memory area in which procedures are created, edited, and loaded. Having a procedure in your workspace merely allows you access to it while working in BASIC09. If you were to leave BASIC09, all procedures in the workspace would be erased.

Nobody wants to retype an entire program every time it is to be run. Consequently, two commands are used to save and recall programs from the OS-9 disk files: **SAVE** and **LOAD**.

The SAVE command can be used in a number of ways:

1. By typing "SAVE" by itself, BASIC09 will create a file with the name of the last edited or run procedure (in this case "myprogram"). If there is already a file with this name, BASIC09 will respond with the prompt: "Rewrite?". If you respond "y" for yes, it will replace the file previously stored in that space with the newly saved procedure. OS-9 will not allow two files with the same name in the same directory. By answering "n" for no, you cancel the SAVE command without affecting the procedure in your workspace or the file in your directory.

2. By typing "SAVE >", followed by a file name, you will save the procedure in a file with the specified name.

3. By typing "SAVE*", followed by a file name, you can save all procedures in your workspace in the specified file.

4. By typing "SAVE" followed by one or more procedures followed by a ">", followed by a file name, you will save the specified procedures in the specified file.

REMINDER: If you exit from BASIC09, your programs WILL NOT automatically be saved. You must make sure to save them using one of the these methods or they will be lost (unless they were saved at some previous time).

RECALLING PROCEDURES

The LOAD command reads in a previously saved program from a disk file. You must give the name of the file with the command:

```
B: LOAD myprogram
```

If you have no procedures currently in your workspace, this command is very simple. If you have a procedure with the same name in your workspace, BASIC09 will replace it with the procedure loaded from the file.

This may cause unnecessary problems. You can rename the file to be loaded before loading it (using the RENAME command), save the file in your workspace under a different name before loading the new procedure or erase the file in your workspace.

At any time, you may permanently erase one or all of your procedures in your workspace by using the KILL command. KILL followed by a procedure name erases the specified procedure. KILL* erases all procedures in the workspace. For example:

```
B: KILL prog1          erases prog1 from workspace
B: KILL*               erases all procedures from workspace
```

While this insures that there will be no conflicts in loading programs with identical names, it is a drastic measure. It should be used only if you are sure there is nothing worth saving.

DIR: LISTING PROCEDURE NAMES IN YOUR WORKSPACE

The "DIR" command causes BASIC09 to display the names and sizes of all procedures in the workspace. This command is used so frequently that there is a quick shorthand for DIR: a simple <cr> when in system mode does the same thing.

The DIR command will display a table of all procedure names with two numbers next to each name. The first column, "proc size", is the size of the corresponding procedure. The "data size" column shows the amount of memory that the procedure requires for its variables.

On the last line, this command shows the amount of free bytes of workspace memory remaining. You can use this information to estimate how much memory your program needs to run. You must have at least as much free memory as the data size of the procedure(s) to be run. If a data size number is followed by a question mark, this means you definitely need more memory.

BASIC09 WORKSPACE AND THE MEM COMMAND

We have talked briefly about the BASIC09 workspace. This is the memory area where procedures are created, edited, and loaded. BASIC09 automatically is allocated approximately 4K bytes of workspace memory from OS-9 when it starts up. There is almost always more memory in the system available. BASIC09 does not take it all so that other tasks running on your computer may have memory also.

If you need more memory, the MEM command can get it if available. Just type MEM and the amount of memory you want (in byte units). Depending on your computer and how it is configured, you can obtain from 4 kilobytes up to nearly 16 megabytes. For example:

```
MEM 20000
```

This command line requests 20,000 (20K) bytes of memory. BASIC09 will always round the amount you request up to the next highest multiple of 256 bytes. If the MEM procedure responds with "What?", the requested amount of memory is not available.

There is another convenient way to request more memory. OS-9 has a "#<memory>" option that lets you specify how much memory to give a program. When you first enter BASIC09 from OS-9, you may specify how much memory you want allocated to BASIC09 by typing "basic09",followed by "#", followed by the amount of memory (in K byte units). For example, to call BASIC09 with 16K of memory:

```
$ basic09 #16K
```

THE DIM STATEMENT: DECLARING VARIABLES

The next simple program we will look at displays another aspect of programming: math and variable manipulation.

First, you should understand the use of variables. Variables are used in procedures to hold values temporarily, during the execution of the procedure. They may change value during the execution. In the example "myprogram", name$ was used as a variable to hold the value: "walden".

There are two ways of declaring a variable: the DIM statement and inferred declaration.

The DIM statement declares the variable name and the type of data that will be assigned to it during the course of the procedure. The DIM statement must occur before the variable is used in the program. This is to prevent a variable from being defined with a default data type (inferred declaration). Consequently, the DIM statement is (by standard convention) used in the first few lines of a procedure.

The syntax for the DIM statement is as follows:

```
DIM <variable> [, <variable>] : <data type>
```

If more than one variable is being declared as the same data type, each variable is separated by a comma. If more than one data type is to be declared in the same DIM statement, each <var>:<type> is separated by a semicolon. For example:

```
DIM x,y,z : integer; a,b,c : real
```

If the DIM statement is not used and variables are present in a procedure, the variables will be declared with default data types. All string variables that are not declared with a DIM statement must end in a dollar sign ($). They are assigned a maximum length of 32 characters.

In the case of "name$" in the example "myprogram", "name$" was declared as a string variable with a length of 32 characters. If the character string assigned to "name$" was longer than 32 characters, only the first 32 would have been accepted and the rest will be lost.

All other variables used will be declared (by default) as "REAL" regardless of the intent of the procedure.

NAMING VARIABLES: RESERVED WORDS

Variables names may be of any length, although it is generally wise to keep them to a manageable length. "This_is_a_legal_variable" would be legal, but tedious to type more than once. Variables names must conform to the following rules:

1. Names must begin with either an underscore or letter.
2. Names can not contain embedded blanks or dollar signs.
3. Names can end in a dollar sign.
4. Names can contain any alphanumeric or underscores.
5. Names may not be any BASIC09 reserved word.

BASIC09 recognizes certain words as "reserved". They may not be used as variable names. These reserved words are all commands and key words used within BASIC09 statements:

BASIC09 RESERVED WORDS

ABS	DIGITS	INPUT	PEEK	SQR
ACS	DIR	INT	PI	SQRT
ADDR	DO	INTEGER	POKE	STEP
AND	ELSE	KILL	POS	STOP
ASC	END	LAND	PRINT	STR$
ASN	ENDEXIT	LEFT$	PROCEDURE	STRING
ATN	ENDIF	LEN	PUT	SUBSTR
BASE	ENDLOOP	LET	RAD	TAB
BOOLEAN	ENDWHILE	LNOT	READ	TAN
BYE	EOF	LOG	REAL	THEN
BYTE	ERR	LOG10	REM	TO
CHAIN	ERROR	LOOP	REPEAT	TRIM$
CHD	EXEC	LOR	RESTORE	TROFF
CHR$	EXITIF	LXOR	RETURN	TRON
CHX	EXP	MID$	RIGHT$	TRUE
CLOSE	FALSE	MOD	RND	TYPE
COS	FIX	NEXT	RUN	UNTIL
CREATE	FLOAT	NOT	SEEK	UPDATE
DATA	FOR	ON	SGN	USING
DATE$	GET	OPEN	SHELL	VAL
DEG	GOSUB	OR	SIN	WHILE
DELETE	GOTO	PARAM	SIZE	WRITE
DIM	IF	PAUSE	SQ	XOR

VARIABLE DATA TYPES

There are five basic data types that BASIC09 recognizes:

INTEGER: Whole numbers (no decimal) ranging from -2,147,483,648 to 2,147,483,647.

REAL: Floating point numbers (decimal point allowed) ranging from (+/-) 2.2 * 10 -308 to 1.8*10 308 (positive or negative). **NOTE:** "*" is the symbol for multiplication. " " is the symbol for exponentiation.

BYTE: Whole numbers (no decimal) ranging from 0 to 255.

STRING: Letters, digits, and/or punctuation.

BOOLEAN: True or False.

NOTE: There are three ways to represent numbers. While REAL numbers appear the most versatile (i.e., they have the greatest range and can represent decimals), math operations involving them are relatively slow. INTEGER and BYTE operations use less memory and are executed faster.

A STRING is a variable length sequence of characters. An "empty" STRING is a special case and contains no characters. A STRING may be declared to have a specified length by using the DIM statement. This may be useful for saving memory space when 32 characters will not be needed (default STRING size is 32 characters).

To declare a STRING length, type "dim", followed by <variable name> : STRING[length]. To declare the variable "word" with a length of 5 characters, for example, you would type:

```
DIM word: STRING[5]
```

The BOOLEAN variable is most often used in conditional statements to divert execution to certain parts of the procedure: If something is true, then do this; otherwise continue.

All five data types are discussed in detail in later chapters of this manual.

CONSTANTS

Constants are frequently used in procedures to assign values to variables. BASIC09 has rules that allow you to specify constants that correspond to the five basic data types.

Numeric Constants

Numeric constants can be either REAL or INTEGER data types. If a number constant includes a decimal point or uses the "E format" exponential form, it forces BASIC09 to store the number in REAL format (even if the number could have been stored in INTEGER or BYTE format). If you specifically want to specify a REAL constant, use a decimal point (for example 12.0). This is sometimes done if all other values in an expression are of type REAL so BASIC09 does not have to do a time-consuming type conversion at run-time.

Numbers that do not have a decimal point but are too large to be represented as integers are also stored in REAL format.

EXAMPLES: 1.0 9.8433218 1.95E+12 10000000000
-.01 -999.000099 -99999.9E-33 5655.34532

Numbers that do not have a decimal point and are in the INTEGER range are treated as INTEGER numbers. BASIC09 will also accept integer constants in hexadecimal in the range 0 to $FFFFFFFF. Hex numbers must have a leading dollar sign.

EXAMPLES: 12 -3000 64000 $20 $FFFE $0

BOOLEAN Constants

The two legal BOOLEAN constants are "TRUE" and "FALSE":

```
DIM flag, state: BOOLEAN
flag := TRUE
state := FALSE
```

STRING Constants

STRING constants consist of a sequence of any characters enclosed by quotation marks. To represent a quotation mark within the string, use two consecutive quotation marks (""). An empty string can also be represented by two consecutive quotation marks.

EXAMPLES:
```
"BASIC09 is a microcomputer language"
""   (a null string)
"An ""older woman"" is wiser"
```

OPERATORS

An operator combines or compares values of operands: constants and variables. Operators (except negation) take two operands and cause some operation to be performed producing a result. This result is generally the same type as the operands. The table on the following page lists the operators available and the types they accept and produce.

Operators have "precedence" which means they are evaluated in a specific order (i.e., multiplication is performed before addition). Parentheses can be used to override natural precedence. Extraneous parentheses, however, may be removed by the compiler. The legal operators are listed below, in order from highest to lowest.

```
NOT     -(negate)       (Highest Precedence)
        **
*       /
+       -
>       <       <>      =       >=      <=
AND
OR      XOR             (Lowest precedence)
```

Operators of equal precedence are shown on the same line, and are evaluated left to right in expressions. The only exception to this rule is exponentiation, which is evaluated right to left. Raising a negative number to a power is not legal in BASIC09.

In the examples below, BASIC09 expressions on the left will be evaluated as indicated on the right. Either form may be entered, but the simpler form on the left will always be generated by the compiler.

```
BASIC09 Representation      Equivalent form

a:= b+c**2/d                a:= b+((c**2)/d)
a:= b>c AND d>e OR c=e      a:= ((b>c) AND (d>e)) OR (c=e)
a:= (b+c+d)/e               a:= ((b+c)+d)/e
a:= b**c**d/e               a:= (b**(c**d))/e
a:= -(b)**2                 a:= (-b)**2
a=b=c                       a:= (b=c)   (returns BOOLEAN value)
```

Operator	Function	Operand Type	Result Type
-	Negation	NUMERIC	NUMERIC
or **	Exponentiation	NUMERIC (positive)	NUMERIC
*	Multiplication	NUMERIC	NUMERIC
/	Division	NUMERIC	NUMERIC
+	Addition	NUMERIC	NUMERIC
-	Subtraction	NUMERIC	NUMERIC
NOT	Logical Negation	BOOLEAN	BOOLEAN
AND	Logical AND	BOOLEAN	BOOLEAN
OR	Logical OR	BOOLEAN	BOOLEAN
XOR	Logical EXCLUSIVE OR	BOOLEAN	BOOLEAN
+	Concatenation	STRING	STRING
=	Equal to	ANY	BOOLEAN
<> or ><	Not equal to	ANY	BOOLEAN
<	Less than	NUMERIC, STRING*	BOOLEAN
<= or =<	Less than or Equal	NUMERIC, STRING*	BOOLEAN
>	Greater than	NUMERIC, STRING*	BOOLEAN
>= or =>	Greater than or Equal	NUMERIC, STRING*	BOOLEAN

* When comparing strings, the ASCII collating sequence is used, so that 0 < 1 < ... < 9 < A < B< ... < Z < a < b< ... < z

NOTE: "NUMERIC" refers to either BYTE, INTEGER or REAL types.

LOOPING STATEMENTS AND A SIMPLE MATH PROGRAM

A simple math program that will print a multiplication table is shown below. In this section we will examine the use of the FOR/TO/NEXT statement and its use in this program.

```
PROCEDURE multable
  DIM a,b : INTEGER
  FOR a=1 TO 4
    FOR b=1 TO 4
      PRINT a; " * "; b; " = "; a*b,
    NEXT b
    PRINT
  NEXT a
END
```

First, enter the editor by typing "e multable":

```
B: e multable
PROCEDURE multable
*
E:
```

Next, type in line by line the above program starting with "DIM a,b : INTEGER". Lower case letters are totally acceptable. The compiler will change all reserved words to upper case letters automatically.

The first line , "DIM a,b : INTEGER", declares the variables a and b as integer data types.

The second line of multable begins the **FOR..TO..NEXT** statement. This type of statement is known as a looping statement. Loops are used to cause repeated or conditional execution of the internal sequence of statements. Generally, loops have one entry at the top and only one exit at the bottom. In this sense, it becomes one statement, regardless of how many individual statements it contains.

The beginning of this loop provides the "loop initialization":

```
FOR a = 1 TO 4
```

This sets the value of "a" initially to 1. It must be paired to the later NEXT statement:

```
NEXT a
```

This increases the value of "a" by 1 (the default step size). If you wanted the control variable to be increased by a different number, such as 3, you must include a STEP declaration on the FOR/TO line:

```
FOR a = 1 TO 4 STEP 3
.
.
NEXT a
```

The STEP declaration may be a positive or negative number. it may be either an integer or a real. If it is a real data type (i.e., STEP .5) the control variable must also be a real data type.

The looping procedure will continue until the control variable is greater than the specified range.

All statements between the matching FOR and NEXT statements will be repeated as long as the control variable is within the specified range of the FOR statement.

Loops can be nested. This allows the internal statements to be executed even more times. In multable the PRINT statement will be executed 4 times for each value of "a". In other words, the nested FOR loop will execute completely (4 times) for each loop through the outer FOR statement.

This feature is at the heart of both computer programming and programming errors. This type of programming allows a small portion of code to execute a large amount of work. It will sometimes (in more complex and more nested loops) come across a special case that will cause the procedure to stop. Good programming foresight should help cut down this type of error. A programmer should know at least what will happen on the first, second, next to the last, and last passes through the looping structure. It is usually these passes where a procedure produces "special case" errors.

Both PRINT statements are worth examining in multable. In the first you will notice that the statement ends in a comma (,). This indicates that the next PRINT statement should be printed on the same line starting at the next tab stop. Whenever a PRINT statement ends in a comma or a semicolon, the next print statement will be written on the same line. Before something will be printed on the next line, a PRINT statement ending with no "," or ";" must be encountered. That is the reason for the second print statement in multable.

The first PRINT statement will print on the same line while the inner loop executes and then will go to the next line when the PRINT statement outside the inner loop is encountered. The second PRINT statement will actually print nothing. It is merely used as a line feed.

If you have correctly entered the multable procedure, you should now be able to run it. First, make sure you are in the command mode. If not, type "q" after the "E:" prompt. Now type "run multable" and your output should look like this:

```
1 * 1 = 1       1 * 2 = 2       1 * 3 = 3       1 * 4 = 4
2 * 1 = 2       2 * 2 = 4       2 * 3 = 6       2 * 4 = 8
3 * 1 = 3       3 * 2 = 6       3 * 3 = 9       3 * 4 = 12
4 * 1 = 4       4 * 2 = 8       4 * 3 = 12      4 * 4 = 16
```

Notice that the beginning of each equation starts every 16 columns. This is the default tab size. When formatting output fields with a comma (,), each field will begin at the next 16 column tab stop.

When making output tables (such as the multiplication table above) it is not always useful to use the default tab stops. There is a **TAB** command that may be used instead. The TAB command syntax is as follows:

```
TAB (column #)
```

This will begin the next output field in the specified column. The column number may also be an evaluated expression. In the multable example, for instance, the PRINT statement could be written as follows to allow only 3 spaces between output fields:

```
PRINT a; " * "; b; " = "; a*b; TAB(b*13);
```

Now the output fields will begin at the start of every thirteen columns.

The WHILE..DO Statement

In the procedure "multable", you were introduced to the FOR..NEXT loop. There are three other types of looping statements: WHILE..DO, REPEAT..UNTIL and LOOP..ENDLOOP.

Any of these three loops could have been used instead of the FOR..NEXT statement in "multable". For example, using the WHILE..DO statement would make the procedure look like this:

```
PROCEDURE multable
  DIM a,b : INTEGER
  a:= 1
  WHILE a < 5 DO
    b:= 1
    WHILE b < 5 DO
      PRINT a; " * "; b; " = "; a*b; TAB(b*13);
      b:= b + 1
    ENDWHILE
    PRINT
  a:= a + 1
  ENDWHILE
  END
```

This "multable" procedure produces exactly the same results as the earlier example.

In this procedure the first difference is noticed in the third and fifth lines. After we have declared "a" and "b" to be integers, we must initialize their values. BASIC09 does not do this for you. It merely assigns a certain space in memory large enough to hold the declared type of data. Consequently, if the variables are not initialized, the results of any type of operation with them will be unreliable.

To initialize a variable, there are two types of assignment statements available. The first is the one used above: the implied assignment. The syntax is as follows:

<variable> := <value or constant>

The second type of assignment statement is the LET statement. It has the exact same syntax, except it begins with the LET command:

LET <variable> := <value or constant>

The actual syntax for the WHILE statement is as follows:

```
WHILE <boolean> DO
   .
   .
ENDWHILE
```

This is a loop construct that tests it's control variable at the top of the loop. <boolean> may be an expression (as in our example) or merely a BOOLEAN variable (i.e., WHILE red DO). The important aspect of this and similar loop constructs is that the control variable must be acted on within the loop. If it is not, the loop will have no exit.

For example, the following loop is an endless loop (because the internal commands do not affect the control variable):

```
a:= 5
WHILE a < 10 DO
   PRINT "This is an endless loop"
ENDWHILE
```

While this example is extremely simple, it is representative of many longer procedures. Sometimes the control variable will be acted upon, but not in a way that affects the control statement. It is important to make sure that there is an exit to every loop construct.

In the preceding "multable" example, the importance of placing the assignment statement inside the first loop can not be overstressed. By this placement, every time the outer loop is executed, the "b" variable is reinitialized. If "b" were initialized outside the first loop (as "a" is), the inner loop would execute the first four times and never again (b> 4).

As you can see, this initialization placement is very important when using loop constructs. It is a common programming error. This is especially true when changing one type of loop construct to another. Again, if you make sure you know what will happen at the beginning and end of each loop sequence, this type of error can be detected and consequently averted.

The REPEAT..UNTIL Statement

Another loop statement is the REPEAT..UNTIL statement. It functions almost identically to the WHILE..DO statement. It's syntax is as follows:

```
REPEAT
   .
   .
UNTIL <boolean>
```

The major difference is that the control variable is tested at the bottom of the loop. This means that the internal commands will be executed at least once (even if the control statement is false to begin with).

The "multable" example would look like this using the REPEAT..UNTIL statement:

```
PROCEDURE multable
  DIM a,b : INTEGER
  a := 1
  REPEAT
    b := 1
    REPEAT
      PRINT a; " * "; b; " = "; a*b; TAB(b*13);
      b:= b + 1
    UNTIL b > 4
    PRINT
    a:= a + 1
  UNTIL a > 4
  END
```

Notice again the initialization of the "b" variable occurs within the first loop.

Because the variable is tested at the bottom of the loop, it usually has an opposite test from one you would find in a WHILE loop. Notice in our WHILE "multable" example the control expression was WHILE b < 5 DO. In the above REPEAT example, it is b > 4.

Again, this is a typical error that occurs when changing one type of loop construct to another in the editing process.

The LOOP..ENDLOOP Statement

The final type of loop construct available in BASIC09 is the LOOP..ENDLOOP statement. It has no built in control statement to test for exit, so it uses an internal construct: the EXITIF..THEN statement. The syntax for these two constructs are as follows:

```
LOOP
   .          EXITIF <boolean> THEN
   .       <     <statements>
   .          ENDEXIT
ENDLOOP
```

As you can see, the LOOP construct would be endlessly executed without the EXITIF construct. The EXITIF construct may be used as many times as needed within a LOOP. In this way, a loop may be exited for many different reasons.

The execution of the loop may be explained by showing an example. The "multable" procedure would look like this:

```
PROCEDURE multable
  DIM a,b : INTEGER
  a := 1
  LOOP
    b:= 1
    LOOP
      PRINT a; " * "; b; " = "; a*b; TAB(b*13);
      b:= b + 1
      EXITIF b > 4 THEN
        PRINT
      ENDEXIT
    ENDLOOP
    a:= a + 1
    EXITIF a > 4 THEN
    ENDEXIT
  ENDLOOP
  END
```

The execution of the above procedure works similar to the REPEAT..UNTIL example used earlier. The inner loop is executed until "b" > 4. Then the statement (PRINT) between the THEN and the ENDEXIT is executed. The control then jumps to the outer loop and executes the first statement possible after the ENDLOOP (a:= a + 1). Notice after the outer loop's control test (EXITIF a > 4 THEN) there is no statement to be executed. This is actually what is known as a null statement (nothing occurs). This is totally acceptable.

CHOOSING THE CORRECT LOOP

As demonstrated, the "multable" example may be written using any of the looping constructs discussed in this chapter. The smallest and least confusing is probably the FOR..NEXT version. This is because we knew exactly how many times it was to execute and exactly how our output was to be formatted.

This will always be the major reason for choosing one looping construct over another. Each loop has its own advantages.

The FOR..NEXT loop is used for procedures that must execute code an exact number of times.

The WHILE..DO loop is used for procedures that test for a control variable before executing any code, and perform only as long as the control statement remains true.

The REPEAT..UNTIL loop is used for procedures that want the code executed at least once regardless of the initial conditions and repeated as long as a control statement remain valid.

The LOOP..ENDLOOP loop can be used for procedures that must test one or more control variables anywhere within the loop (and perhaps more than once).

There is one more consideration in choosing a correct loop for your procedure. The EXITIF..THEN construct may be used in any of the looping constructs to exit anywhere within the loop. This allows much greater freedom in constructing your procedures.

CONDITIONAL CONTROL: THE IF..THEN STATEMENT

One of the most used control statements in programming is the conditional statement: IF..THEN..ELSE. The syntax is as follows:

```
IF <boolean> THEN
   <statement>
ELSE
   <statement>
ENDIF
```

This allows certain statements to be executed only if specified conditions exist. The following example demonstrates the use of the IF..THEN..ELSE statement:

```
PROCEDURE MESSAGE
   PRINT "Type your name."
   INPUT name$
   PRINT "Would you like the message of the day, ";name$;"? (y/n)"
   INPUT answer$
   IF answer$ = "y" THEN
     PRINT "Space is the future"
   ELSE
     PRINT "Suit yourself."
   ENDIF
   PRINT "Bye, "; name$; ".  It's been nice talking to you."
   END
```

As you can see one of two messages will be printed depending on the input from the user of this procedure. Of course the condition could be dependent upon a computed value just as easily. And there could be many statements or procedures separating the THEN and ELSE segments of the conditional. This example simply shows how execution can be diverted temporarily to a set of alternate statements.

The IF..THEN statement may be used in one other method:

```
IF <boolean> THEN <line#>
```

This sends the control of the procedure to the specified line number if the control condition is met. It is recommended to rarely use the IF..THEN statement in this way.

WARNING: Multiple ELSE statements are not considered errors by the compiler (they will not generate error signals), however they will produce irregular and non-reliable results.

CHANGING YOUR PROCEDURE: THE EDITOR

Once you have written a procedure, you may want to (or have to) change it. There are a number of commands available in edit mode for this express purpose. They are:

<return>	Moves the edit pointer forward one line.
+[<number>]	Moves the edit pointer forward the specified number of lines (default is 0).
+*	Moves the edit pointer forward to the end of the procedure.
-[<number>]	Moves the edit pointer back the specified number of lines.
-*	Moves the edit pointer to the beginning of the procedure.
<space> <text>	Inserts a line directly before the line the current edit pointer is on.
<space> <line#> <text>	Inserts or replaces numbered line.
<line#> <return>	Moves edit pointer to specified line.
c[*]<del><string><del><string><del>	Replaces the first string with the second string in the line the edit cursor in on. If used with an "*", the entire procedure will be searched and replaced. The delimiters are any matching character that is not within either string. For example:

```
c.why.why not?.        legal syntax
c?why?why not??        illegal syntax
```

d[*] [line#]	Deletes the specified line. If no line is specified, then the current line is deleted. If a negative number is specified, the specified number of lines before the current line will be deleted. "d -*" or "d +*" is also allowed. They delete all lines before or after the current line respectively. "d*" deletes the entire procedure.
l[*] [<number>]	Lists the specified number of lines from the current edit pointer. If the number is negative, the specified number of lines before the current line is displayed. "l*" lists the entire procedure.

q Quits the editor.

r[*] [<number>],[<increment>] Renumbers the numbered lines in a procedure. The "r" command begins at the current line and renumbers the first numbered line found with the specified number. After that it increments the line number by the specified increment. If an "*" is used then the renumbering begins with the start of the procedure. This will also renumber any references to line numbers (GOTO or IF..THEN statements). The default values for renumbering are a starting value of 100 with an increment of 10.

s[*] <del> <string> <del> Searches for the indicated string on the current line. If an "*" is used the entire procedure is searched and the pointer will be moved to the first matching string. Delimiters (<del>) follow the same rules as in the "c" command.

<escape> Quits the editor.

To use the editor, you must be in edit mode. Using the procedure, "multable", we can examine some of the edit mode commands. First change to edit mode by typing "e multable" after the "B: " prompt.

You will notice that the first line of the procedure is displayed with an asterisk (*) before the line. This tells you that the edit pointer is positioned at this line.

Type "l*" to review the entire procedure and you will see that there is an asterisk (still) in front of the first line. The edit pointer has not changed position.

The first thing that should be added is a header. This should be added just before the WHILE structure and after the initializations.

Type "+3" or hit the <return> key three times. This will move the edit pointer forward three lines to the line that begins with "WHILE...". By typing in an appropriate PRINT line and hitting a <return>, the entered line will be placed directly before the edit pointer:

```
PRINT TAB(13) "Multiplication Tables"
```

The edit pointer will again be displayed as pointing to the WHILE statement (as it hasn't moved). Type "PRINT" to further format your output. This will double space between the header and the body of the output.

Another change that would be nice is to double space between lines of output. Type "+6" (move ahead 6 lines) and the edit pointer will be moved down 6 lines. Type "PRINT" here.

If a larger table were to be printed, the output would have to be reformatted further. To print a multiplication up to ten, type: "c".5.10." This changes all occurrences of "5" to "10". There is only one problem, not all terminals can support 130 character lines.

There is a BASIC09 feature that takes care of this type of formatting nicely: the **POS** function. Insert the following lines of code after "PRINT a; ..." by positioning the edit pointer on the b:=b+1 line and typing:

```
IF POS > 55 THEN PRINT
ENDIF
```

This tells the compiler to start a new line when the character position is farther than 55 after the preceding PRINT statement.

Now your "multable" procedure should look like this:

```
PROCEDURE multable
  DIM a,b : INTEGER
  a := 1
  PRINT TAB(13); "Multiplication Tables"
  PRINT
  WHILE a < 10 DO
    b:= b +1
    WHILE b < 10 DO
      PRINT a; " * "; b; " = "; TAB(b*13);
      IF POS > 55 THEN PRINT
      ENDIF
      b:= b + 1
    ENDWHILE
    PRINT
    PRINT
    a:= a + 1
  ENDWHILE
  END
```

LINE NUMBERS AND THE GOTO STATEMENT

As mentioned before, the listing BASIC09 displays is not in the exact format as was input. There is a space between the I-code address and the actual procedure itself. This is reserved for line numbers.

Line numbers are required in many versions of BASIC (but not BASIC09). Line numbers must be positive whole numbers in the range of 1 to 32767. They do not need to be used for every line. They are generally used in conjunction with a GOTO or GOSUB statement.

The GOTO statement transfers control unconditionally to the specified line. For example:

```
PROCEDURE gotodemo
  PRINT "This is a GOTO example"
  GOTO 10
  PRINT "This line will never be printed"
  10 PRINT "It works but it's dangerous"
  END
```

As you can see, a GOTO statement could cause certain parts of a procedure's code to be excluded from execution. There are generally much better ways of obtaining the same results without ever using a GOTO statement.

The use of the GOSUB statement will be discussed in detail in chapter 2.

A STRONG RECOMMENDATION: If you do not have to use line numbers or the GOTO statement, DON'T!! Your procedures will be shorter, faster and easier to edit. There will be less chance of error. If you must use a GOTO statement use it sparingly and always document the code with a comment.

PUTTING IT ALL TOGETHER

With the various control statements and editing commands presented in this chapter, (with a little practice) you should be able to write some fairly advanced procedures.

As an example of this, consider the following program. It combines the first two programs shown in this chapter while integrating the other commands presented:

```
PROCEDURE mathtables
  DIM less:BOOLEAN
  DIM answer$:STRING[1]
  DIM a,b,c :INTEGER
  PRINT "Type your name"
  INPUT name$
  PRINT "Hi, "; name$; "!  ";
  PRINT "Would you like to print a multiplication table?"
  PRINT "Type y for yes.  Type any other key for no."
  INPUT answer$
  IF answer$="y" THEN
    PRINT "What is the number you want to multiply?"
    PRINT "(please specify a number between 1 and 100)"
    INPUT a
    PRINT "What is the range of the multiplication table?"
    PRINT "(type 2 numbers between 1 and 50 separated by a space)"
    INPUT b,c
    WHILE b=c DO
      PRINT "Please specify two different numbers for the range."
      INPUT b,c
    ENDWHILE
    PRINT "Thank you, "; name$
    count=1
    REPEAT
      PRINT b; " * "; a; " = "; b*a,
      IF POS > 55 THEN PRINT
      ENDIF
      IF b>c THEN b:=b-1
        less=FALSE
      ELSE
        less=TRUE
        b:=b+1
      ENDIF
    UNTIL b=c+1 AND less OR b=c-1 AND NOT(less)
  ELSE PRINT "Your loss, "; name$
  ENDIF
  END
```

USER NOTES

ARRAYS

An array is an ordered sequence of data types. An array may be one, two or three dimensional. A one dimensional array is called a vector. A two dimensional array is called a table. A three dimensional array is called a matrix.

The size of an array will depend on the number of elements in each dimension and the size of each element. The array size is declared with a DIM statement.

The vector array, "Names", has 80 STRING elements, with each element being 30 characters. It is declared as follows:

```
DIM Names(80) : STRING [30]
```

Multiple dimension arrays are declared similarly. For example the table array, "Phonebook", has 2 dimensions: 80 rows in 5 columns of STRING elements, each element being 30 characters. This totals 400 elements. It is declared as follows:

```
DIM Phonebook(80,5): STRING [30]
```

Like variables, when arrays are declared they are not initialized to null values. They will contain random values that could give both unsuspected and quite annoying results. It is therefore always good programming practice to initialize an array before loading values into it.

The following procedure initializes "Phonebook" to null values:

```
PROCEDURE init
  DIM Phonebook(80,5): STRING [30]
  DIM x,y : INTEGER
  FOR x := 1 TO 80
    FOR y := 1 TO 5
      Phonebook(x,y) = ""
    NEXT y
  NEXT x
  END
```

Numeric array elements should be initialized to 0 in the same manner.

Once an array is initialized, it can be loaded in various ways. The simplest is to assign individual elements values. This is done by an assignment statement referencing the specified element:

```
Phonebook(1,1) := "Robert Doggett"
Phonebook(1,2) := "unlisted"
```

This is simple but it is extremely time consuming. By using the looping structures available and the input statement we can load the array interactively:

```
PROCEDURE Addphone
  DIM Phonebook(80,5):STRING[30]
  DIM x,y:INTEGER
  DIM done:BOOLEAN
  DATA "NAME","PHONE","ADDRESS","CITY, STATE","ZIP CODE"
  done:=FALSE
  x:=1
  WHILE NOT(done) AND x<=80 DO
    FOR y=1 TO 5
      READ prompt$
      PRINT "ENTER "; prompt$
      INPUT Phonebook(x,y)
    NEXT y
    INPUT "If finished, type ""done"", otherwise <return>.",flag$
    IF flag$="done" THEN
      done:=TRUE
    ELSE
      x:=x+1
    ENDIF
  ENDWHILE
  END
```

In this procedure, the DATA and READ statements are introduced. The READ statement reads sequentially from the DATA statements output list. When the list is exhausted, it will start reading from the beginning of the list again. In this case, "Addphone" uses the READ statement to change prompts for each element of the array.

Notice the INPUT statement following the FOR..NEXT structure:

```
INPUT "If finished, type ""done"", otherwise <return>.",flag$
```

This INPUT statement prints out the character string instead of the regular "?" prompt. This eliminates an extra PRINT statement.

THE TYPE DECLARATION

While the "Addphone" procedure is adequate for storing names and phone numbers, an array of varying data types and sizes would be more efficient. In the "Phonebook" array each element is 30 characters long, regardless of the use of the field. A phone number field would rarely need to be greater than 20 digits. A zip code could be held in 9.

By use of the TYPE declaration, BASIC09 allows the creation of user defined data types. A user defined data type may consist of any of combination of the 5 basic data types, arrays and other user defined data types. For example:

```
TYPE rec=street:STRING[30];cityst:STRING[30];zip:STRING[9]
TYPE ENTRY=name:STRING[30]; phone:STRING[20]; address:rec
DIM Phonebook(80) : ENTRY
```

The above example is functionally the same array as in "Addphone". However, it is quite smaller and can be accessed much easier. By labeling each field with a descriptive name, there is no doubt as to what should be stored there.

Each field within an element is referred to by the array name and the number of the element, followed by a period, followed by the field name (or names each separated by a period):

```
Phonebook(32).address.zip := 50311
```

EXTERNAL FILES

Now that we can define and access variable length records, we have to be able to save them for future use. The current Addphone program will only hold the memory of the phone number while it is running.

To save the input information, we must open and put information into an external file.

There are two types of files that OS-9 supports:

SEQUENTIAL FILES: These files hold records containing ASCII characters. There can be any number of characters within a record. There can be any number of records in a file (within the limit of your memory on disk). Records are separated by a carriage return.

RANDOM ACCESS FILES: These files contain records that store data in the same manner as BASIC09; in binary. There are no carriage returns to indicate the end of a record. Records must be of fixed size. There may be any number of records in the file (within limits of your disk).

Sequential files are generally used for text files. The reason for this is the way in which data is accessed.

To store data in a sequential file BASIC09 supports two commands: **READ** and **WRITE**. The WRITE command sends each character in the record to the file and then sends a carriage return to indicate the end of record. The READ command reads characters from a file until it finds a carriage return.

Sequential files must be read (sequentially) one record at a time, starting from where the file pointer is positioned. Because of the way data is stored in sequential files, the only way to position the file pointer is to read or write a record.

If you would like to access the third record of a sequential file, you must either read the first two records or rewrite them to place the file pointer at the beginning of the third record. If you were to rewrite the first record with a new record that is larger than the original, you would also be writing over the beginning of the second record. Obviously, this could cause many problems.

To effectively edit sequential files, a text editor (word processor, etc.) must be used.

To store data in a random access file, BASIC09 supports two commands: **GET** and **PUT**. The PUT command puts the record in your file in a specified place. The GET command retrieves the record.

Data elements in a random access file can be accessed individually. The BASIC09 functions, **SEEK** and **SIZE**, can be used together to find and place the file pointer at the beginning of any element. Through proper use of these functions, you can GET the record you want or PUT any record where you want it.

Creating Files

Before you can access a file, you must first create it. The syntax of the **CREATE** statement is as follows:

CREATE #<int or byte var>, "<path>": <access mode>

This statement creates the specified file (<path>) and opens a path number to access it. The path number is assigned to the variable by the create statement. It must be either an INTEGER or BYTE data type. The access mode may be any of the following:

WRITE **UPDATE** **EXEC**

The WRITE access mode will only allow you to send (WRITE/PUT) data to the file. The UPDATE access mode will allow you to both send and receive data (WRITE/PUT and READ/GET). The EXEC mode is used when you need to store machine language code to be executed. It is rarely used except by advanced BASIC09 programmers.

When a file is created, it has a length of zero. It will expand automatically as you send data to it.

The following procedure creates a random access file to hold "Phonebook" records. It assigns the prompted values to the individual "Phonebook" data fields one at a time. It puts the entire data structure, however, into the file at one time.

The PUT and the GET statement have the same syntax. PUT/GET, followed by the path number variable, followed by the data structure desired.

One final important note, when you open a path to a file, you must close it. Notice the line at the end:

CLOSE #file

The CLOSE statement will close any path number given it. It is prudent to never close a path that you didn't open, unless absolutely required.

```
PROCEDURE Phonebook
  TYPE rec=street:STRING[30];cityst:STRING[30];zip:STRING[9]
  TYPE ENTRY=name:STRING[30]; phone:STRING[20]; address:rec
  DIM Phonebook : ENTRY
  DIM file : INTEGER
  CREATE #file, "phonebook": UPDATE
  PRINT "Enter the information asked for by the prompt"
  PRINT "If information not available, hit <return>"
  REPEAT
    INPUT "name:            ", Phonebook.name
    INPUT "phone number:    ", Phonebook.phone
    INPUT "street address: ", Phonebook.address.street
    INPUT "city, state:     ", Phonebook.address.cityst
    INPUT "zip code:        ", Phonebook.address.zip
    PUT #file, Phonebook
    PRINT "If finished, type ""done""."
    INPUT "Otherwise hit the <return>  ", done$
  UNTIL done$ = "done"
  CLOSE #file
  END
```

While this works well if the "Phonebook" file does not yet exist, this is not a procedure for adding or accessing records within the "Phonebook" file. Trying to CREATE an already existing file will cause an error (#218).

To access an already existent file, you must use the OPEN statement. The OPEN statement has exactly the same syntax as the CREATE statement:

```
OPEN #<int or byte var>,"<path>": access mode
```

There are five access modes that the OPEN statement uses:

```
READ     WRITE     UPDATE     EXEC     DIR
```

READ allows you to read data from a file. WRITE allows you to write data from a file. UPDATE allows you to read and write to a file. EXEC looks for and stores your file in your execution directory. DIR opens a directory for read only.

You can OPEN a file in more than one mode (if a valid combination). For example:

```
OPEN #file, "example": READ + EXEC
```

NOTE: "DIR +" either WRITE or UPDATE will cause an error. "READ + WRITE" is the same as UPDATE.

Using A Random Access File

To illustrate the access capabilities of random access files, examine the procedure on page 2-9. It is based on the previous "Phonebook" procedure.

Note the use of the BASIC09 statement, ON ERROR GOTO. This gives control to the specified line if an error occurs. In this case it is used to bypass the problem of re-creating an existent file:

```
    flag = TRUE
    ON ERROR GOTO 100
    CREATE #file, "phonebook": UPDATE
    flag = FALSE
100 IF flag THEN OPEN #file, "phonebook": UPDATE
    ENDIF
```

Normally, when an error occurs, the procedure will terminate and BASIC09 will change to the DEBUG mode. A FALSE value is assigned to "flag" directly between the CREATE and IF..THEN OPEN structure. This assures that only one of the two access statements (OPEN or CREATE) will be executed.

The ON ERROR without the GOTO effectively turns off the previous ON ERROR statement. This is important, because any other error that might occur would otherwise be routed to line 100.

Because this procedure can be used to both read and write to the "phonebook" file, it is imperative to access the file in UPDATE mode.

By prompting for one of three conditions (add, access or done), control can be channelled to any of these conditions. By placing this control structure within a REPEAT loop, the user may both add entries and access entries in the same session. By adding REPEAT loops within "add" and "access" sections of the code, the user may add or access as many records as desired.

The key statement that allows this access is the SEEK statement. In the "add" loop, the file pointer is moved to the end of file:

```
SEEK #file, FILSIZ (#file)
```

The SEEK command will position the file pointer directly after the number of bytes specified. The FILSIZ function returns the size of the file specified by its path. Consequently by using both together, the file pointer is positioned at the end of file.

In the "access" loop, the SEEK command positions the file pointer at the beginning of the user-specified record. For the sake of simplicity, the method used to specify records in this procedure is by number (i.e., first record, second record). The BASIC09 function, SIZE, returns the size of the specified data structure. By multiplying this size by one less than the desired record, the file pointer will be correctly positioned:

```
SEEK #file, SIZE (Phonebook) * (record - 1)
```

```
PROCEDURE Phonebook
  TYPE rec=street:STRING[30];cityst:STRING[30];zip:STRING[9]
  TYPE ENTRY=name:STRING[30]; phone:STRING[20]; address:rec
  DIM Phonebook : ENTRY
  DIM file,record : INTEGER
  DIM flag : BOOLEAN
  flag = TRUE
  ON ERROR GOTO 100
  CREATE #file, "phonebook": UPDATE
  flag = FALSE
100 IF flag THEN OPEN #file, "phonebook": UPDATE
  ENDIF
  ON ERROR
  REPEAT
    PRINT "Would you like to add or access entries?"
    PRINT "Or are you finished for now?"
    INPUT "Type ""add"", ""access"" or ""done"".", answer$
    IF answer$ = "add" THEN
      SEEK #file, FILSIZ (#file)
      PRINT "If information not available, hit <return>"
      REPEAT
        INPUT "name:           ", Phonebook.name
        INPUT "phone number:   ", Phonebook.phone
        INPUT "street address: ", Phonebook.address.street
        INPUT "city, state:    ", Phonebook.address.cityst
        INPUT "zip code:       ", Phonebook.address.zip
        PUT #file, Phonebook
        PRINT "If finished, type ""done""."
        INPUT "Otherwise hit the <return>  ", done$
      UNTIL done$ = "done"
    ELSE
      IF answer$ = "access" THEN
        REPEAT
          INPUT "What number record would you like?", record
          SEEK #file, SIZE (Phonebook) * (record - 1)
          GET #file, Phonebook
          PRINT "name:           ", Phonebook.name
          PRINT "phone number:   ", Phonebook.phone
          PRINT "street address: ", Phonebook.address.street
          PRINT "city, state:    ", Phonebook.address.cityst
          PRINT "zip code:       ", Phonebook.address.zip
          PRINT "If finished, type ""done""."
          INPUT "Otherwise hit the <return>  ", done$
        UNTIL done$ = "done"
      ENDIF
    ENDIF
  UNTIL answer$ = "done"
  CLOSE #file
```

SUBROUTINES

If the previous program appears unnecessarily complex, it is because it really is. When a certain portion of a procedure has a specific function that is used repeatedly, it is generally separated from the main procedure by placing it in a subroutine. Some code would become more clear if separate from the main procedure. Likewise, it too can be put in a subroutine.

Subroutines are generally used to clarify code or to avoid repeated use of the same code throughout a procedure. In the previous program, both the "add" and the "access" loop could be placed in subroutines to improve clarity.

Subroutines are accessed by the GOSUB statement. The syntax for this structure is as follows:

```
        GOSUB <line#>

<line#> (subroutine)
        RETURN
```

Control is unconditionally passed to the specified line by the GOSUB statement. BASIC09 will continue to execute sequentially from that point until RETURN is encountered. Control is then returned to the line immediately following the GOSUB statement.

BASIC09 supports a related statement that provides better use of subroutines: ON..GOSUB. The syntax for this statement is as follows:

```
        ON <integer expr> GOSUB {<line#>, <line#>}

<line#> (subroutine)
        RETURN
<line#> (subroutine)
        RETURN
```

ON..GOSUB evaluates the <integer expression> and transfers control to the corresponding line number in the list following GOSUB. If the integer is greater than the number of line numbers in the list, no subroutine is executed. When used with a previous input statement, ON..GOSUB can be used to run menu-type subroutines.

Note the changes in the "Phonebook" procedure, by using subroutines:

```
   PROCEDURE Phonebook
     TYPE rec=street:STRING[30];cityst:STRING[30];zip:STRING[9]
     TYPE ENTRY=name:STRING[30]; phone:STRING[20]; address:rec
     DIM Phonebook:ENTRY; flag:BOOLEAN; file,answer,record:INTEGER
     flag := TRUE
     ON ERROR GOTO 100
     CREATE #file, "phonebook": UPDATE
     flag := FALSE
100  IF flag THEN OPEN #file, "phonebook": UPDATE
     ENDIF
     ON ERROR
     REPEAT
       PRINT "TYPE: ""1"" for add new entries"
       PRINT "      ""2"" for access previous entries"
       PRINT "      ""3"" for exit procedure"
       INPUT  answer
       ON answer GOSUB 200,300
     UNTIL answer = 3
     CLOSE #file
     END

200  SEEK #file, FILSIZ (#file)
     REPEAT
       PRINT "If information not available, hit <return>"
       INPUT "name:              ",Phonebook.name
       INPUT "phone number:      ",Phonebook.phone
       INPUT "street address:    ",Phonebook.address.street
       INPUT "city, state:       ",Phonebook.address.cityst
       INPUT "zip code:          ",Phonebook.address.zip
       PUT #file,Phonebook
       PRINT "If finished, type ""done""."
       INPUT "Otherwise hit the <return>  ",done$
     UNTIL done$="done"
     RETURN

300  REPEAT
       INPUT "What number record would you like?",record
       SEEK #file,SIZE(Phonebook)*(record -1)
       GET #file,Phonebook
       PRINT "name:              ",Phonebook.name
       PRINT "phone number:      ",Phonebook.phone
       PRINT "street address:    ",Phonebook.address.street;
       PRINT "city, state:       ",Phonebook.address.cityst
       PRINT "zip code:          ",Phonebook.address.zip
       PRINT "If finished, type ""done""."
       INPUT "Otherwise hit the <return>  ",done$
     UNTIL done$="done"
     RETURN
```

CALLING PROCEDURES

Many times certain procedures could be used (like subroutines) by many different procedures. BASIC09 allows procedures to call each other (and even call themselves!).

The RUN statement is used to call an external procedure. BASIC09 will look first in the workspace, then your data directory and finally your execution directory. If it is found outside your workspace, BASIC09 will load and then run it.

The RUN statement can include parameters (a list of values) to be passed to the called procedure. The called procedure in turn must have a PARAM statement with variables of the same size and data type as the values passed. Parameters may be any type of data structure.

If a parameter is a constant or an expression, it is passed "by value". This means it can be changed by the called procedure, but the changes will not be returned to the calling procedure.

If a parameter is a variable, array, or data structure, it is passed "by reference". This means that any changes made to the value of the parameter will be returned to the calling procedure. BYTE data types may only be passed by reference.

The syntax for the PARAM statement is as follows:

```
PARAM <declaration sequence>: <type>
```

Items in the declaration sequence are separated by commas. More than one type may be declared on the same line. For example:

```
PARAM a,b: INTEGER; c,d: REAL; listing(10): BYTE
```

The following example procedures pass "by reference" an array of integers. "Example" creates the array and passes it to "prin". "Prin" prints the array and passes it back to "example". "Example" passes it to "reverse" which reverses the order of the array. The reversed array is now passed back to "example". "Prin" is run once more to validate the reversal process.

```
PROCEDURE example
  DIM a,intlist(10): INTEGER
  FOR a = 1 TO 10
    intlist(a) = a
  NEXT a
  RUN prin(1,10,intlist)
  RUN reverse(1,10,intlist)
  RUN prin(1,10,intlist)
  END
```

```
PROCEDURE prin
  PARAM a,b,prin(10): INTEGER
  DIM c: INTEGER
  FOR c = a TO b
    PRINT prin(c);" ";
  NEXT c
  PRINT
  END
```

```
PROCEDURE reverse
  PARAM a,b,intlist(10):INTEGER
  DIM c, temp(10): INTEGER
  FOR c = b TO a STEP -1
    temp(b+1-c) = intlist(c)
  NEXT c
  intlist = temp
  END
```

The output should look like this:

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
```

FORMATTED OUTPUT: THE PRINT USING STATEMENT

BASIC09 has a powerful output editing capability useful for report generation and other applications where formatted output is required. The output editing uses the **PRINT USING** statement:

SYNTAX: **PRINT** [<path#>] **USING** <str expr> , <output list>

The string expression is evaluated and used as a "format specification". This contains specific formatting directives for each item in the "output list". BLANKS ARE NOT ALLOWED IN FORMAT STRINGS! The <path#> is optional and can redirect the output to the corresponding device.

The items in the output list can be constants, variables, or expressions of any basic type. As each output item is processed, it is matched up with a specification in the format list. The type of each expression result must be compatible with the corresponding format specification. If there are fewer format specifications than items in the output list, the format specification list is repeated again from its beginning as many times as necessary.

A format string has one or more format specifications which are separated by commas. There are two kinds of specifications: ones that control output editing of an item from the output list and ones that cause an output function by themselves (such as tabbing and spacing). There are six basic output editing directives. Each has a corresponding one-letter identifier:

R	real format
E	exponential format
I	integer format
H	hexadecimal format
S	string format
B	boolean format

The letter is followed by a positive constant number called the "field width". This number indicates the exact number of columns in which the output is to be printed. It must allow for the data **AND** "overhead" character positions such as sign characters, decimal points, exponents, etc. The field width must be between 1 and 255.

Some formats have additional mandatory or optional parameters that control subfields or select editing options. Real and exponential formats use the "fraction field" to specify the number of digits to the right of the decimal point. The fraction field is separated from the field width by a decimal point. For example, a field width of 10 and a fraction field of 6 is represented by "10.6".

All formats can use the "justification" option, which specifies whether the output is to be centered, left or right justified within the output field. Fields are commonly right-justified in reports because it arranges them into columns with decimal points aligned in the same position. The symbols used in the justification specifications are: < (left), > (right), (center).

In the previous "prin" procedure, the following print statement was used:

```
PRINT prin(c);" ";
```

The extra space was added to separate the integers being printed. A similar result can be obtained by using the PRINT..USING statement:

```
PRINT USING "I3>",prin(c);
```

This will format each integer in a 3 space print field (I3). The integers will be right justified (>). Some caution must be taken when formatting. While a leading sign is not printed (when positive), space for it must be allowed. "I3>" will only print 2 digit integers.

A full explanation of the PRINT.. USING statement and each of the format types can be found in the reference section of this manual.

end of chapter 2

USER NOTES

GENERAL EXECUTION PERFORMANCE OF BASIC09

The BASIC09 multi-pass compiler produces a compressed and optimized low-level "I-code" for execution. Compared to other BASIC languages, program storage is greatly decreased and execution speed is increased.

High-level language interpreters have a general reputation for slowness which is probably not deserved. Because the BASIC09 I-code is kept at a very powerful level, a single, fast, I-code interpretation will often result in many MPU instruction cycles (such as execution of floating-point arithmetic operations). Thus, for complex programs, there is little performance difference between execution or I-code and straight machine-language instructions. This is generally not the case with traditional BASIC interpreters that have to "compile" from text as they run or even with "tokenized" BASICs that must perform table-searching during execution. BASIC09 I-code instructions that reference variable storage, statements, labels, etc., contain the actual memory addresses, so no table searching is ever required. Of course, BASIC09 fully exploits the power of the 68000's instruction set which was optimized for efficient execution of compiler-produced code.

Because the BASIC09 I-code is interpreted, a variety of entry-time and run-time tests and development aids are available to help in program development: aids not available on most compilers. The editor reports errors immediately when they are entered, the debugger allows debugging using the original program source statements and names, and the I-code interpreter performs run time error checking of things such as array bound errors, subroutine nesting, arithmetic errors, and other errors that are not detected (and usually crash) native-compiler-generated code.

OPTIMUM USE OF NUMERIC DATA TYPES

BASIC09 includes several different numeric representations (i.e., REAL, INTEGER, and BYTE) and does "automatic type conversions" between them. It is easy to write expressions or loops that take at least ten times longer to execute than is necessary. Some particular BASIC09 numeric operators (+, -, *, /) and control structures (FOR..NEXT) include versions both for REAL and INTEGER values. The INTEGER versions, of course, are much faster, and may have slightly different properties (e.g., INTEGER divides discard any remainder). Type conversions take time, so expressions whose operands and operators are of the same type are more efficient.

Nonetheless, INTEGER operations are faster because they generally have corresponding 68000 machine-language instructions. Overall program speed will increase and storage requirements will decrease if INTEGERs are used whenever possible. INTEGER arithmetic operations use the same symbols as REAL but BASIC09 automatically selects the INTEGER operations when working with an integer-value result. Only if all operands of an expression are of types BYTE or INTEGER will the result also be INTEGER.

Sometimes, similar or identical results can be obtained in a number of different ways at various execution speeds. For example, if the variable "value" is an integer, then "value*2" will be a fast integer operation. However, if the expression is "value*2." the value "2." will be represented as a REAL number, and the multiplication will be a REAL multiplication which will also require that the variable "value" will have to be transformed into a REAL value, and finally the result of the expression will have to be transformed back to an INTEGER value if it is to be assigned to a variable of that type. Thus a single decimal point will slow this particular operation down by about ten times!

LOOPING QUICKLY

When BASIC09 identifies a FOR..NEXT loop structure with an INTEGER loop counter variable, it uses a special integer version of the FOR..NEXT loop. This is much faster than the REAL-type version and is generally preferable. Other kinds of loops will also run faster if INTEGER type variables are used for loop counters.

When writing program loops, remember that statements INSIDE the loop may be executed many times for each single execution OUTSIDE the loop. Thus, any value which can be computed before entering a loop will increase program speed.

OPTIMUM USE OF ARRAYS AND DATA STRUCTURES

BASIC09 internally uses INTEGER numbers to index arrays and complex data structures. If the program uses subscripts that are REAL type variables or expressions, BASIC09 has to convert them to INTEGER before they can be used. This takes additional time, so use INTEGER expressions for subscripts whenever you can.

Note that the assignment statement (LET) can copy identically sized data structures. LET is much faster than copying arrays or structures element-by-element inside a loop.

THE PACK COMMAND

The PACK command produces a compressed version of a BASIC09 procedure. Depending on the number of comments, line numbers, etc., programs will execute from 10% to 30% faster after being .PACKed. Minimizing use of line numbers will even speed up procedures that are unPACKed.

ELIMINATING CONSTANT EXPRESSIONS AND SUB-EXPRESSIONS

Consider the expression:

```
x := x+SQRT(100)/2
```

It is exactly the same as the expression:

```
x := x+5
```

The sub-expression "SQRT(100)/2" consists of constants only, so its result will not vary regardless of the rest of the program. But every time the program is run, the computer must evaluate it. This time can be significant, especially if the statement is within a loop. Constant expressions or sub-expressions should be calculated by the programmer while writing the program (using DEBUG mode or a pocket calculator).

FAST INPUT AND OUTPUT FUNCTIONS

Reading or writing data a line or record at a time is much faster than one character at a time. Also, the GET and PUT statements are much faster than READ and WRITE statements when dealing with disk files. This is because GET and PUT use the exact binary format used internally by BASIC09. READ, WRITE, PRINT and INPUT must perform binary-to-ASCII or ASCII-to-binary conversions which take time.

PROFESSIONAL PROGRAMMING TECHNIQUES

One sure way to make a program faster is to use the most efficient algorithms possible. There are many good programming "cookbooks" that explain useful algorithms with examples in BASIC or PASCAL. Thanks to BASIC09's rich vocabulary you can use algorithms written in either language with little or no adaptation.

CHAPTER 3
PROGRAM OPTIMIZATION

BASIC09 eliminates any possible excuse for not using good structured programming style that produces efficient, reliable, readable and maintainable software. BASIC09 generates optimized code to be executed by the 68000 which is the most powerful 16-bit processor in existence at the time of this writing. A computer can only execute what it is told to execute. No language implementation can make up for an inefficient program. An inefficient program is evidence of a lack of understanding of the problem. The result is likely to be hard to understand and hard to update if program specifications change (they always do). The identification of efficient algorithms and their clear, structured expression is indicative of professionalism in software design and is a goal in itself.

end of chapter 3

PART II - THE BASIC09 REFERENCE GUIDE

- System Mode
- Edit Mode
- Execution Mode
- Debug Mode
- Data Structures and Variables
- Expressions, Operators, Functions
- Program Statements and Structure
- Input and Output Statements

INTRODUCTION TO COMMAND MODES

BASIC09'S FOUR MODES

Any time you are working with BASIC09, you will be working in one of four modes:

SYSTEM MODE: In this mode, system commands are executed.

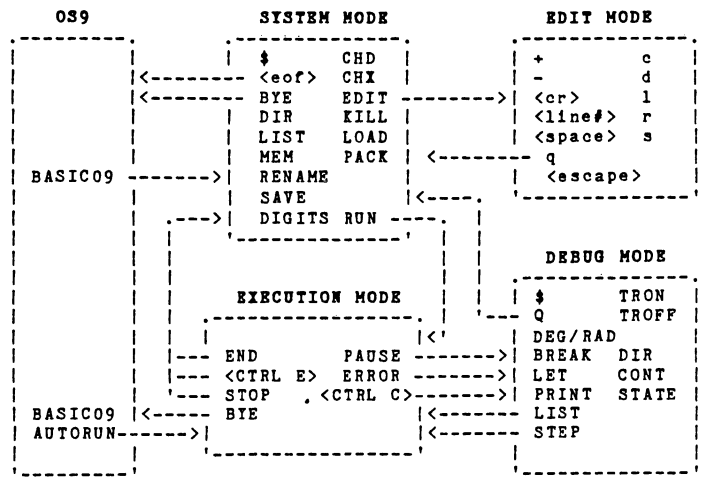
EDIT MODE: In this mode, procedures are created or edited.

EXECUTION MODE: In this mode, procedures are run.

DEBUG MODE: In this mode, procedures are tested for errors.

Certain commands in each mode will change what mode BASIC09 is in. The following is a graphic representation of which commands will accomplish this and what mode they are carried out in.

BASIC09 MODE CHANGES



SYNTAX NOTATION USED IN DESCRIPTIONS OF SYSTEM COMMANDS

Individual descriptions of the commands available in each mode follow. In order to precisely describe their formats, the syntax notation shown below is used.

[ ]	items in brackets are optional
{ }	items in braces can be optionally repeated
<procname>	a procedure name
<pathlist>	an OS-9 file name
<number>	a decimal or hex number

SYSTEM MODE COMMANDS

System Mode includes commands to save, load and examine procedures; commands to interact with OS-9; and other commands to control the workspace environment. A complete list of system commands is given below.

System Mode Commands

$	BYE	CHD/CHX	DIGITS	DIR	E/EDIT	KILL
LIST	LOAD	MEM	PACK	RENAME	RUN	SAVE

The system commands are processed by the BASIC09 "command interpreter" which always identifies itself with the "B:" prompt. It is entered automatically when BASIC09 is started up and whenever you exit any other mode. Commands can be entered in either upper or lower-case letters.

Commands such as DIR, MEM, "$" and BYE do not operate on specific procedures but may have optional or required parameters. Commands such as SAVE, LOAD, PACK, KILL and LIST can be made to operate on a specific procedure or on ALL procedures within the workspace.

If the command is used with a specific procedure name, the command is applied to only that procedure. This example will (only) display the procedure named "pete":

```
LIST pete
```

The asterisk is a special name that means "all procedures in the workspace". Therefore, if the command is given followed by an asterisk it is applied to all procedures. This example will display all of the procedures in the workspace:

```
LIST*
```

If the command is given without any name at all, the "current working procedure" is used, which means the name of the procedure last given in another command. The DIR command prints an asterisk before the current procedure's name so it can be found at any time.

If you have not yet given a name in any command, the name "PROGRAM" is automatically used. Some commands require a file name as well as (one or more) procedure names. They usually require that a ">" precede the file name so it is not mistaken for a procedure name. If you omit the file name, the name of the (first) procedure is used instead.

NOTE: In this manual, the phrase "file name" means an OS-9 "pathlist" which can describe either a file or device.

Here are some examples:

```
SAVE tom,bill >myfile
SAVE* big_file
SAVE tic,tac,toe            (same as SAVE tic,tac,toe >tic)
```

Another class of commands uses only one procedure name, or the current working name if a name is omitted. These commands change the mode of BASIC09 by exiting the command mode and entering another mode. These commands are:

RUN enters Execution Mode to run a procedure.

EDIT enters Edit Mode to create or change a procedure.

NOTE: Debug Mode can not be entered directly from the system mode

$
"Shell" Command

SYNTAX: $ [<text>]

FUNCTION: This command calls the OS-9 Shell command interpreter to process an OS-9 command or to run another program. Running the Shell command does not cause BASIC09 or its workspace to be disturbed.

If the "$" is followed by text, the Shell is called to process the text as a single OS-9 command line. After the command is executed, BASIC09 is immediately reentered.

If no text is given, BASIC09 is suspended and the OS-9 Shell is called to process multiple command lines individually entered from the keyboard. Control is returned to BASIC09 when an end-of-file character (usually ESCAPE) is entered. The contents of the BASIC09 workspace is not affected. This is a convenient way to temporarily leave BASIC09 to manipulate files or perform other "housekeeping".

This command is the "gateway" to OS-9 from inside BASIC09. It allows access to any OS-9 command or to other programs. It also permits creation of concurrent processes and other real-time functions.

EXAMPLES:

B: $copy file1 file2	calls the OS-9 "copy" command
B: $r68 sourcefile&	calls the assembler as a background task
B: $basic09 fourier(20)&	starts another concurrent BASIC09 program

BYE (or <eof> character)
Exit BASIC09

FUNCTION: BYE exits BASIC09 and returns to OS-9 or the program that called BASIC09. Any procedures in the workspace are lost if not previously saved. The end-of-file character (usually the escape key) does the same thing.

CHD / CHX
Change Directories

SYNTAX: CHD <pathlist>
CHX <pathlist>

FUNCTION: CHD changes the current OS-9 user Data Directory to the specified pathlist which must be a directory file. BASIC09 uses the Data Directory to LOAD or SAVE procedures.

CHX changes the current OS-9 user Execution Directory to the specified pathlist which must be a directory file. The Execution Directory is used to PACK or auto-load packed modules.

EXAMPLE: CHD /d1/joe/games

DIGITS
Formats Numerical Output (Real Numbers)

SYNTAX: DIGITS [<number>]

FUNCTION: DIGITS controls the number of digits that are printed when REAL numbers are output.

DIGITS also controls the precision of transcendental calculation. The minimum precision is 1 digit to the right of the decimal point, and the maximum precision is 15. If the result is not within this range (1 to 15 precision), the result is brought into range without error. When no <number> is specified, DIGITS will display the current precision.

EXAMPLE:
```
PROCEDURE Digitsdemo
DIM x: REAL
DIGITS 2
INPUT x
PRINT x
END

RUN Digitsdemo
? 44.9234692
44.
```

DIR
Display Directory of Workspace

SYNTAX: DIR [<pathlist>]

FUNCTION: DIR displays the name, size and variable storage requirement of each procedure presently in the workspace. The current working procedure has an asterisk before its name. All "packed" procedures have a dash before their name (see PACK). The available free memory within the workspace is also given. If a pathlist is given, output is directed to that file or device.

A question mark next to a data storage size means the workspace does not have enough free memory to run that procedure.

NOTE: This command should not be confused with the OS-9 "DIR" command. They have completely different functions.

EDIT / E
Enter Edit Mode

SYNTAX: EDIT [<procname>]
E [<procname>]

FUNCTION: EDIT (E) exits system mode and enters edit mode. If the procedure named does not exist, a new one is created.

See Chapter 5 for a complete description of how edit mode works.

EXAMPLES: E newprog

EDIT newprog

KILL / KILL*
Delete Procedure From Workspace

SYNTAX: KILL [<procname> {,<procname>}]
KILL*

FUNCTION: KILL deletes the procedure(s) specified by <procname>. KILL* clears the entire workspace. This process may take some time if there are many procedures in the workspace.

EXAMPLES: KILL formulas

KILL prog1, prog2, prog7

LIST / LIST*
Displays Listing of Procedure

SYNTAX: LIST [<procname> [,<procname>]] [> <pathlist>]
LIST* [<pathlist>]

FUNCTION: LIST prints a formatted listing of one or more procedures. LIST* prints a formatted listing of all procedures in the workspace. The listing includes the relative I-code storage addresses in hexadecimal format in the first column. The second column is reserved for program line numbers (if line numbers are used).

If a pathlist is given, the listing is output to that file or device. This option is commonly used to print hard-copy listings of programs.

LIST prints on the OS-9 Standard Error Path (#2) if no pathlist is given.

IMPORTANT NOTE: If an "*" is used, the file name (<pathlist>) follows immediately WITHOUT a ">" before it!

EXAMPLES: LIST* /p

LIST prog2,prog3 >/p

LIST prog5 >temp

LOAD
Load Procedure Into Workspace

SYNTAX: LOAD <pathlist>

FUNCTION: LOAD loads all procedures from the file specified into the workspace. As procedures are loaded, their names are displayed. If any of the procedures being loaded have the same name as a procedure already in the workspace, the existing procedures are erased and replaced with the procedure being loaded.

If the workspace fills up before the last procedure in the file is loaded, an error (#32) is given. In this case, not all procedures may have been loaded, and the one being loaded when the workspace became full may not be completely loaded. The user should KILL the last procedure, use the MEM command to get more memory or KILL unnecessary procedure(s) to free up space, then LOAD the file again.

EXAMPLE: LOAD quadratics

MEM
Display or Request Workspace Memory

SYNTAX: MEM <number>

FUNCTION: MEM used without a number displays the present total workspace size in (decimal) bytes. If a number is given, BASIC09 asks OS-9 to expand or contract the workspace to that size. A hex value can be used if preceded by a dollar sign. If MEM responds with "What?", you either asked for more memory than is available, tried to give back too much memory (there has to be enough to store all procedures in the workspace), or gave an invalid number.

EXAMPLE: MEM 18000

PACK / PACK*
Pack Procedure

SYNTAX: PACK [<procname> [,<procname>]] [> <pathlist>]
PACK* [<pathlist>]

FUNCTION: PACK causes an extra compiler pass on the specified procedure(s). This removes names, line numbers, non-executable statements, etc. The result is a smaller, faster procedure(s) that CAN NOT be edited or debugged but can be executed by BASIC09 or by the BASIC09 run-time-only program called "RunB".

If a pathlist is not given, the name of the first procedure in the list will be used as the file name. The packed file will be stored in your execution directory.

The procedure is written to the file/device specified in OS-9 memory module format suitable for loading in ROM or RAM OUTSIDE the workspace. Basic09 will automatically load the packed procedure when you try to run it later on. Here is an example sequence that demonstrates packing a procedure:

```
PACK sort       packs procedure "sort" and creates a file
KILL sort       kills procedure inside the workspace
RUN sort        run (sort will be loaded outside workspace)
KILL sort       done; we delete "sort" from outside memory
```

The last step does not have to be done immediately if you will be using the procedure again later, but you should kill it whenever you are done so its memory can be used for other purposes.

PACK (continued)

WARNING: THE PACKED FILE CAN NOT BE LOADED INTO THE WORKSPACE LATER ON. Always perform a regular SAVE before packing a procedure!

EXAMPLES: PACK proc1,proc2 >packed.programs

RENAME
Renames a Procedure

SYNTAX: RENAME <procname>,<new procname>

FUNCTION: RENAME changes the name of a procedure. It can be used to allow two copies of the same procedure in the workspace under different names.

EXAMPLE: RENAME thisproc thatproc

RUN
Execute a Procedure

SYNTAX: RUN [<procname> [(<expr> , [<expr>])]]

FUNCTION: RUN executes the specified procedure. Technically speaking, BASIC09 then leaves System Mode and enters Execution Mode.

A parameter list can be used to pass expected parameters to the procedure. This is generally used in the same way that a RUN statement inside a procedure calls another procedure. The only restriction is that all parameters must be constants or expressions without variables. See the PARAM statement description.

The procedure called can be normal or "packed". If the procedure is not found inside BASIC09's workspace, BASIC09 will call OS-9 to attempt to LINK to an external (outside the workspace) module. If this fails, BASIC09 attempts to LOAD the procedure from a file of the same name.

EXAMPLES: RUN getdata

RUN invert("the string to be inverted")

RUN power(12,354.06)

RUN power($32, sin(pi/2))

SAVE / SAVE*
Write Procedure to an Output File

SYNTAX: **SAVE** [<procname> [,<procname>] [> <pathlist>]]
SAVE* [<pathlist>]

FUNCTION: SAVE writes the procedure(s) (or all procedures with SAVE*) to an output file or device in source format. This command is similar to the LIST command except the output is not formatted and I-code addresses are not included. If a pathlist is not specified, it will default to the name of the first procedure listed.

If a file of the same name already exists, SAVE will prompt with:

rewrite?

You may answer "Y" for yes which causes the existing file to be rewritten with the new procedure(s); or "N" to cancel the SAVE command.

IMPORTANT NOTE: If an "*" is used, the file name (<pathlist>) follows immediately WITHOUT a ">" before it!

EXAMPLES: **SAVE** proc2,proc3,proc4 >monday.work

SAVE* newprogram

SAVE

SAVE >testprogram

end of chapter 4

USER NOTES

EDIT MODE COMMANDS

Edit Mode ("The Editor") is used to create or modify BASIC09 procedures. It is entered from System Mode by the EDIT (or E) command. As soon as the Edit Mode is entered, the prompt "E:" will be displayed. If you have used a text editor before, you will find the BASIC09 editor similar to many others except for these two differences:

1. The editor is both "string" and "line number" oriented. The use of line numbers is optional and text can be corrected without re-typing the entire line.

2. The editor is interfaced to the BASIC09 compiler and "decompiler" which lets BASIC09 do continuous syntax error checking and permits programs to be stored in memory in more compact compiled form.

The Editor includes the following commands. Each command is described in detail later in this chapter.

<cr>	move edit pointer forward one line
+[<number>]	move edit pointer forward
-[<number>]	move edit pointer backward
<space> <text>	insert unnumbered line
<space> <line#> <text>	insert or replace numbered line
<line#> <cr>	find numbered line
c	change string
d	delete line
l	list line(s)
q	quit editing
r	renumber line
s	search for string
<esc>	quit editing

HOW THE EDITOR WORKS

BASIC09 programs are always stored in memory in a compiled form called "I-code" (short for "Intermediate Code"). I-code is a complex binary coding system for programs that lies in between your original "source" program and the computer's native "machine language". I-code is relatively compact, can be executed rapidly, and most importantly, can be reconstructed almost exactly back to the original source program. The Editor is closely connected to the "compiler" and "decompiler" systems within BASIC09 that translate source code to I-Code and vice-versa.

Whenever you enter (or change) a program line and hit "return", the compiler instantly translates this text to the internal "I-code" form. Whenever BASIC09 needs to display program lines back, it uses the decompiler to translate the I-code back to the original "source" format. These processes are completely automatic and do not require any special action on your part.

This technique has several advantages. First, it allows the text editor to report many (syntax) errors immediately so you can correct them instantly. Secondly, the I-code representation of a program is more compact (by about 30%) than its original form so you can have larger programs in any given amount of available memory.

When programs are listed by BASIC09, it is possible that they will have a slightly different appearance than the way they were originally typed in, but they will always be functionally identical to the original form. A different appearance can happen if the original program had extraneous spaces between keywords, unnecessary parentheses in expressions, etc. BASIC09 keywords are always automatically capitalized.

When you have finished editing the procedure, use the "q" (for "quit") command to exit the Edit Mode and return to the System Mode. When you give the "q" command, the compiler performs another "pass" over the entire procedure. At this time, syntax that extends over multiple lines is checked and errors reported. Examples of these kinds of errors are: GOTO or GOSUB to a non-existent line, missing variable or array declarations, improperly constructed loops, etc. These errors are reported using an error code and the hexadecimal I-code address of the error. For example:

```
01FC ERR #000:043
```

This message means that error number 43 was detected in the line that included I-code address 01FC (hexadecimal). The LIST command gives the I-code addresses so you can locate lines with errors reported during the compiler's second pass.

LINE-NUMBER ORIENTED EDITING

The editor has the capability to work on programs with or without line numbers. Line numbers must be positive whole numbers in the range of 1 to 32767.

If you have experience with another version of the BASIC language, this is the kind of editing you probably used. However, well structured programs seldom really need line numbers. If you don't have to use line numbers, don't. Your programs will be shorter, faster, and easier to read.

The line number oriented commands are:

<space> <line#> <text>	insert or replace numbered line
<line#> <cr>	find numbered line
r	renumber line
r*	renumber all lines

To enter or replace a numbered line, simply type in a <space>, followed by the line number and statement. Numbered lines can be entered in any order but will be automatically stored in ascending sequence. To move the edit pointer to a numbered line, type the line number followed by a carriage return. The editor will move to that line (or the line with the next higher number if the specified number is not found) and display it. The line may be deleted using the "d" command.

The "r" renumber command will uniformly resequence all numbered lines and lines that refer to numbered lines. The syntax for this command is as follows:

```
r [ <beg line #>  [,<incr> ] ] <CR>
r*[ <beg line #>  [,<incr> ] ] <CR>
```

The first format renumbers the program starting at the current line forward. Lines are renumbered using <beg line#> as the initial line number. <incr> is added to the previous line number for the next line's number. The following example will give the first line number 200, the second 205, etc. If <beg line#> and/or <incr> are not specified, the values 100 and 10, respectively, are assumed.

```
r 200,5
```

The second form of the command is identical except it renumbers all lines in the procedure.

STRING-ORIENTED EDITING

Most editor commands are string-oriented, which means that you can enter or change whole or partial lines without using line numbers at all. You will find that string-oriented editing is generally faster and more convenient.

Because line numbers are not used, there must be another way to directly specify lines on which to work. To do this, the editor maintains an "edit pointer" that indicates which line is the present working location within the procedure. String oriented commands work relative to this point.

The editor shows you the location of the edit pointer by displaying an "*" at the left side of the program line where the edit pointer is presently located.

Moving the Edit Pointer

The "+" and "-" commands are used to reposition the edit pointer:

-	moves backward one line
- <number>	moves backward <number> of lines
-*	moves to the beginning of the procedure
+	moves forward one line
+ <number>	moves forward <number> of lines
+*	moves to the end of procedure

The <number> indicates how many lines to move. Backward means towards the first line of the procedure. If the number is omitted, a count of one is used (this is true of most edit commands).

A line consisting of a carriage return only also moves the pointer forward one line, which makes it easy to "step" through a program a line at a time. Therefore, the following commands all do the same thing:

```
<CR>
+ <CR>
+1 <CR>
```

Inserting Lines

The Insert Line function consists of a "space" followed by a BASIC09 statement line. The statement is inserted just ahead of the edit pointer position (the space itself is not inserted).

Deleting Lines

The "d" command is used to delete one or more lines. Its format is:

```
d [<number>] <CR>
d*
```

The first form deletes <number> lines starting at the current edit pointer location. If the number is negative, that many lines BEFORE the current line are deleted. If a line number is omitted, only the current line is deleted.

The second form deletes ALL lines in the procedure (caution!). The editor accepts "d+*" and "d-*" to mean delete to the end or the beginning of the procedure respectively.

Listing Lines

The "l" command is used to display one or more lines. It also has the forms:

```
l [<number>] <CR>
l*
```

The first form will display <number> lines starting at the current edit pointer position. If the number is NEGATIVE, previous lines will be listed. The second form displays the entire procedure. Neither changes the edit pointer's position. The line that is the present position of the edit pointer is displayed with a leading asterisk.

Search: Finding STRINGS

A string is a sequence of one or more characters. This can include letters, numbers or punctuation in any combination. Strings are very useful because they allow you to change or locate just part of a statement without having to type the whole thing.

In the Editor, strings must be surrounded by two matching characters (called **delimiters**). This allows the editor to know where the string begins and ends. The characters used for delimiters are not considered part of the string and therefore can not also appear within the string.

Strings used by the Editor should not be confused with BASIC09's data type which is also called **STRING** - they are quite different.

The "s" command may be used to locate the next occurrence or all occurrences of a string. The format for this command is:

```
s <delim> <match str> [<delim>] <cr>
s*<delim> <match str> [<delim>] <cr>
```

The first format searches for the <match str> starting with the line the current edit pointer is on. If any line at or following the edit pointer includes a sequence of characters that match the search string, the edit pointer is moved to that line and the line is displayed. If the string can not be located, the following message will be displayed and the edit pointer will remain at its original position:

```
CAN'T FIND: "<match str>"
```

The "s*" variation searches for all occurrences of the string in the procedure starting at the present edit pointer and displays all lines in which it is found. The edit pointer is moved to the last line where the string occurred.

Here are some examples:

```
E:s/counter/                looks for:  counter
E:s.1/2.                    looks for:   1/2

E:s?three blind mice?       looks for:   three blind mice
```

Change: STRING Substitution

The change string function is a very handy tool that can eliminate a tremendous amount of typing. It allows strings within lines to be located, removed and replaced by another string. This command is commonly used for fixing lines with errors without having to retype the entire line or changing a variable name throughout a program. The format for the "c" command is:

```
c <delim> <match str> <delim> <repl str> [<delim>] <CR>
c*<delim> <match str> <delim> <repl str> [<delim>] <CR>
```

In the first form, the editor looks for the first occurrence of the match string starting at the present edit pointer position. If found, the match string is removed from the line and the replacement string inserted in its place.

The second form works the same way but changes ALL occurrences of the match string in the procedure starting at the present edit pointer position.

The "c*" command will stop anytime it finds or creates a line with an error.

WARNING: Sometimes you can inadvertently change a line you did not intend to change because the match string is imbedded in a longer string. For example, if you attempt to change "no" to "yes" and the word "normal" occurs before the "no" you are looking for, "normal" will change to "yesrmal"!

EXAMPLES: c/xval/yval/

```
c*,GOSUB 5300,GOSUB 5500
```

end of chapter 5

USER NOTES

RUNNING PROGRAMS

To run a BASIC09 procedure, enter:

```
RUN <procname>
```

If the procedure you want to run was the last procedure edited, listed, saved, etc., you can execute it without giving a procedure name at all (the "*" shown in the DIR command identifies this procedure).

If the procedure expects parameters, they can be given on the same command line. They must all be constant numbers or strings, as appropriate, and must be given in the correct order. For example:

```
RUN add(4,7)
```

This is used to call a program (such as the one that follows) and pass to it the specified parameters.

```
PROCEDURE add
PARAMETER a,b          a,b will receive the values 4,7
PRINT a+b
END
```

The ability to pass parameters to a program allows you to specifically initialize program variables. Sometimes certain procedures are parts of a larger software system and are designed to be called from other procedures. You can use this feature to individually test such procedures by passing them test values as parameters.

The RUN statement causes BASIC09 to enter Execution Mode, causing the procedure to run until one of these things happen:

1. An END or STOP statement is executed.
2. You type CONTROL-E
3. A run-time error occurs
4. You type CONTROL-C (keyboard interrupt)

NOTE: In cases 1 and 2, you will return to System Mode. In cases 3 and 4 you will enter Debug Mode.

EXECUTION MODE: TECHNICALLY SPEAKING

The RUN statement is simple and normally you do not need to know what is happening inside BASIC09 when you use it. The technical description of Execution Mode that follows is given for the benefit of advanced BASIC09 programmers.

Execution mode is BASIC09's state when any procedure is being run. It involves execution of the I-code of one or more procedures inside or outside the workspace. Many procedures can be in use because they are able to call each other (or themselves) and "nest" exactly like subroutines do.

Execution Mode can be entered in two ways:

1. By means of the RUN system command.

2. By BASIC09's auto-run feature.

The Auto-run feature allows BASIC09 to get the name of a file to load and run from the same command line used to call BASIC09. The file loaded and run can be either a SAVED file (in the data directory), or a PACKED file (in the execution directory). The file may contain several procedures; the one executed is the one with the same name as the file. Parameters may be passed following the pathname specified. When using the auto-run feature, upon finishing execution, control will return to BASIC09's comand mode. For example, the following OS-9 command lines use this feature:

```
$ BASIC09 printreport("Past Due Accounts")

$ BASIC09 evaluate(COS(7.8814)/12.075,-22.5,129.055)
```

end of chapter 6

OVERVIEW OF DEBUG MODE

What is **symbolic debugging**? Simply stated, it is testing and manipulation of programs using the actual names and program statements used in the program. This is accomplished by BASIC09's powerful symbolic debugging commands:

$	BREAK	CONT	DEG/RAD	DIR
LET	LIST	PRINT	Q	STATE

This chapter will discuss how the Debug Mode can let you watch your program run in slow motion. This allows you to observe each statement as it is executed. This chapter also includes how to use the Debug Mode as a powerful calculator.

Debug Mode is entered from Execution Mode in one of three ways:

1. When an error occurs during execution of a procedure (that is not intercepted by an **ON ERROR GOTO** statement within the program).

2. When a procedure executes a **PAUSE** statement.

3. When a keyboard interrupt (**CONTROL C**) occurs.

When any of the above happen, Debug Mode announces itself by displaying the suspended procedure name like this:

```
BREAK: PROCEDURE test5
D:
```

Notice that Debug Mode displays a "D:" prompt when it is awaiting a command. Any Debug Mode commands can then be used to examine or change variables, turn trace mode on/off, etc. Depending on which commands are used, execution of the program can be terminated, resumed or executed one source line at a time.

DEBUG MODE COMMANDS

$
Shell Command

SYNTAX: $ <text>

FUNCTION: "$" calls OS-9's Shell command interpreter to run a program or OS-9 command. This command executes exactly the same as the System Mode "$" command.

BREAK
Set Breakpoint

SYNTAX: BREAK <proc name>

FUNCTION: BREAK sets up a "breakpoint" at the procedure named. This command is used when procedures call each other and provides a way to re-enter Debug Mode when returning to a specific procedure.

To illustrate how BREAK works, suppose there are three procedures in the workspace: PROC1, PROC2 and PROC3. Assume that PROC1 calls PROC2 which in turn calls PROC3. While PROC3 is executing, you type CONTROL-C to enter debug mode. To use the BREAK command, type:

```
D: BREAK proc1
ok
D:
```

Notice that BREAK responds with "ok" if the procedure was found on the current RUN stack. If you wish you can use the STATE command to verify that the three procedures are indeed "nested" as expected.

Now, you can resume execution of PROC3 by typing CONT. After PROC3 terminates, control passes back to PROC2, which eventually returns to PROC1. As soon as this happens, the breakpoint you set is encountered, PROC1 is suspended, and Debug Mode is reentered.

There are three characteristics of BREAK you should note:

1. The breakpoint is removed as soon as it occurs.

2. You can use one breakpoint for each active procedure.

3. You can not put a breakpoint on a procedure unless it has been called but not yet re-entered. Hence, BREAK can not be used on procedures that have not yet run.

CONT
Continue Execution

FUNCTION: CONT causes program execution to continue at the next statement. It may be used to resume programs suspended by CONTROL-C, PAUSE statements, BREAK command breakpoints or after non-fatal run-time errors.

DEG / RAD
Select Degree or Radian Units For Computation

FUNCTION: The DEG and RAD commands tell the system (by setting a state flag) that either degrees or radians, respectively, is to be used as the angle unit measure used by trigonometric functions. These commands only affect the procedure currently being debugged or run.

DIR
Display Workspace Directory

SYNTAX: DIR [<pathname>]

FUNCTION: DIR displays workspace procedure directory in exactly the same way as the System Mode DIR command.

LET
Assignment Statement

SYNTAX: LET <var> := <expr>

FUNCTION: The Debug Mode LET command is essentially the same as the BASIC09 LET program statement. It allows the value of a procedure variable to be set to a new value using the result of the evaluated expression. The variable names used in this command must be the same as in the original "source" program, otherwise an error is generated. LET does not work on user-defined data structures.

LIST
List Current Procedure

FUNCTION: LIST displays a formatted source listing of the suspended procedure with I-code addresses. An asterisk is printed to the left of the statement where the procedure is suspended. Only the current procedure may be listed.

PRINT
Print Present Value of Variables

SYNTAX: PRINT [#<expr>,] [USING <expr>,] <expr list>

FUNCTION: PRINT can be used to examine the present value of variables in the suspended program. All variable names must be the same as in the original program. No new variable names can be used. User-defined data structures can not be printed.

Q
Quits Debug Mode

FUNCTION: Q terminates execution of all procedures and exits Debug Mode by returning to System Mode. Any open paths are closed at this point.

STATE
List Calling Order of Procedures

FUNCTION: STATE lists the calling ("nesting") order of all active procedures. The highest-level procedure will always be shown at the bottom of the calling list, and the lowest-level procedure will always be listed first.

EXAMPLE:
```
D:state
PROCEDURE DELTA
CALLED BY BETA
CALLED BY ALPHA
CALLED BY PROGRAM
```

STEP
Single (or Specified) Line Execution

SYNTAX: STEP [<number>] or <CR>

FUNCTION: STEP allows the suspended procedure to be executed one or more source statements at a time.

For example, "STEP 5" would execute the equivalent of the next 5 source statements. A debug command line which is just a carriage return is considered the same as "STEP 1". The STEP command is most commonly used with the trace mode on. This allows the original source lines to be seen as they are executed.

NOTE: Because compiled I-code contains actual statement memory addresses, the "top" or "bottom" statements of loop structures are usually executed just once. For example, in FOR...NEXT loops the FOR statement is executed once, so the statement that appears to be the "top" of the loop will actually be the one following the "FOR" statement.

TRON / TROFF

FUNCTION: These commands turn the suspended procedure's trace mode on and off. In trace mode, the compiled code of each equivalent statement line is reconstructed to source statements and displayed before the statement is executed. If the statement causes the evaluation of one or more expressions, an equal sign and the expression result(s) are displayed on the following line(s).

Trace mode is local to a procedure. If the suspended procedure calls another, no tracing occurs until control returns to the calling procedure (unless the called procedure has trace mode on).

DEBUGGING TECHNIQUES

If your program does not do what you expect it to, it is sure to show one of two symptoms: premature termination due to an error or incorrect results.

The first case will automatically send you into Debug Mode. In the second case, you have to force the program into Debug Mode either by hitting CONTROL-C (assuming you have time to do so), or by using Edit Mode to put one or more PAUSE statements in the program. Once you are in Debug Mode you can bring its powerful commands to bear on the problem.

Usually the first step after an error stops the program is to use the PRINT command to look at the present values of crucial program variables. Bad values are usually quite apparent. Perhaps you forgot to initialize a variable or forgot to increment a loop counter.

If examining variables is not fruitful, the next step is to place a PAUSE statement at the beginning of the suspect procedure or at a place within the code where you think things begin to go amiss. Then rerun the program. When the program hits the PAUSE statement, it enters the DEBUG mode.

Next, turn the trace mode on and actually watch your program run. Type: D: **TRON**

Having done this, hit the carriage return key once for every statement you want to trace. You will see the original source statement, and if expressions are evaluated by the statement, Debug Mode will print an equal sign and the result of the expression.

Notice that some statements such as FOR and PRINT may cause more than one expression to be evaluated.

Using this technique you can watch your program run one step at a time until you see where it goes wrong.

If in the process of tracing, you encounter a loop that works OK but executes 200 statements repetitively, you do not have to trace line by line. In this case, you may turn the trace off and use the STEP command to quickly run through the loop. Then turn trace mode back on and resume single-step debugging. The command sequence for this is:

```
D: TROFF
D: STEP 200
D: TRON
```

REMINDER: Trace mode is "local" to one procedure only. If the procedure being tested returns to another procedure you will need to use the BREAK command or put a PAUSE statement in the procedure to enter Debug Mode. If you call another procedure from the procedure being debugged, tracing will stop when it is called until it returns. If you want to trace the called procedure as well, it will need its own PAUSE statement.

DEBUG MODE AS A DESK CALCULATOR

The simple program listed below turns Debug Mode into a powerful desk calculator. It's function is simple: "Calculator" declares 26 working variables then goes into Debug Mode so you can use interactive PRINT and LET statements.

```
PROCEDURE Calculator
DIM a,b,c,d,e,f,g,h,i,j,k,l,m
DIM n,o,p,q,r,s,t,u,v,w,x,y,z
PAUSE
END
```

Recall that while in Debug Mode you can not create new variables, hence the DIM statements that pre-define 26 working variables for you. If you wish you can use more or fewer variables. The PAUSE statement causes Debug Mode to be entered. Here's a sample session:

```
B: run calculator
BREAK: PROCEDURE Calculator
D:let x:=12.5
D:print sin(pi/2)
.707106781
D:let y:=exp(4+0.5)
D:print x,y
12.5    90.0171313
D:Q
B:
```

REMINDER: The Debug Mode PRINT command can use PRINT USING to produce formatted output (including hexadecimal).

end of chapter 7

USER NOTES

CHAPTER 8
DATA TYPES, VARIABLES AND DATA STRUCTURES

DATA TYPES

A computer program's primary function is to process data. The performance of the computer, and even sometimes whether or not a computer can handle a particular problem, depends on how the software stores data in memory and operates on it. BASIC09 offers many possibilities for organizing and manipulating data.

Complicating matters somewhat is the fact that there are many types of data. Data can be numbers used for counting or measuring or textual data composed of letters, punctuation, etc. These different data types can seldom be mixed. Not only do they have different storage size requirements, but they are logically incompatable. For example, it would be meaningless to multiply letters and punctuation.

Even within the same general kind of data, it is frequently advantageous to have different ways to represent data. BASIC09 lets you choose from three different ways to represent numbers, each having its own advantages and disadvantages. The decision to use one depends entirely on the specific program you are writing.

In order for you to select the most appropriate way to store data variables, BASIC09 provides five different basic data types. BASIC09 also lets you create new customized data types based on combinations of the five basic types.

DATA STRUCTURES

A **data structure** refers to storage for more than one data item under a single name. Data structures can be composed of various data types. Data structures are often the most practical and convenient way to organize large amounts of similar data.

The simplest kind of data structure is the **array**, which is a table of values. The table has a single name, and the storage space for each individual value is numbered. Arrays are created by DIM statements.

For example, to create an array having five storage spaces called "AGES", we can use the statement:

```
DIM AGES(5):INTEGER
```

"(5)" tells BASIC09 how many spaces to reserve. ":INTEGER" indicates the array's data type. To assign a value of 22 to the third storage space in the array we can use the statement:

```
LET AGES(3):=22
```

THE FIVE BASIC DATA TYPES

BASIC09 includes five basic data types: **BYTE**, **INTEGER**, **REAL**, **STRING** and **BOOLEAN**. The first three types are used to represent numbers. The STRING type is used to represent character data, and the BOOLEAN type is used to represent the logical values of either TRUE or FALSE.

Arrays of any of these data types can be created using one, two or three dimensions. The table below gives an overview of the characteristics of each type:

BASIC DATA TYPE SUMMARY

Type	Allowable Values	Memory Requirement
BYTE	Whole Numbers 0 to 255	One byte
INTEGER	Whole Numbers -2,147,483,648 to 2,147,483,647	Four bytes
REAL	Floating Point (+/-) 2.2*10 -308 to 1.8*10 308	Eight bytes
STRING	Letters, digits, punctuation	One byte/char
BOOLEAN	True or False	One byte

REAL numbers appear to be the most versatile data type. They have the greatest range and are floating-point. Arithmatic operations involving them, however, are relatively slow (by a factor directly related to the memory required) when compared to the INTEGER or BYTE types.

Therefore, using INTEGER values for loop counters, indexing arrays, etc. can significantly speed up your programs. While the BYTE type is not appreciably faster than INTEGER, it conserves memory space in some cases and is very useful as a building block for complex data types in other cases.

If you neglect to specify the type of a variable, BASIC09 will automatically assume the REAL data type.

The BYTE Data Type

BYTE variables hold integer values in the range 0 through 255 which are stored as a single byte. BYTE values are always converted to INTEGER values and/or REAL values for computation, thus they have no speed advantage over other numeric types. However, BYTE variables require only 1/4 the storage used by integers, and 1/8 that used by reals.

Attempting to store an integer value outside the BYTE range to a BYTE variable will result in storage of the least-significant 8-bits (the value modulo 256) without error.

The INTEGER Data Type

INTEGER variables consist of four bytes of storage and hold a numeric value in the range -2,147,483,648 through 2,147,483,647 as signed 32-bit data. Decimal points are not allowed. INTEGER constants may also be represented as hexadecimal values in the range $00000000 through $FFFFFFFF to facilitate address calculations. INTEGER values are printed without a decimal point. INTEGER arithmetic is faster and requires less storage than REAL values.

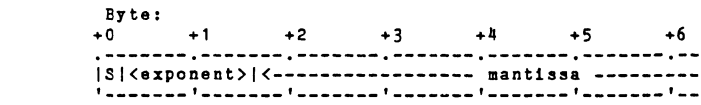
Arithmetic which results in values outside the INTEGER range does not cause run-time errors but instead "wraps around" modulo 4,294,967,296; (i.e., 2,147,483,647 + 1 yields - 2,147,483,648). Division of an integer by another integer yields an integer result, and any remainder is discarded. Values outside the INTEGER range are converted to REAL values and consequently will return an input error when passed to a procedure as integers. Additionally, certain functions (**LAND**, **LNOT**, **LOR**, **LXOR**) use integer values but produce results on a non-numeric bit-by-bit basis.

The REAL Data Type

The REAL data type is the default type for undeclared variables. However, a variable may be explicitly typed REAL (e.g., twopi:REAL) to improve a program's internal documentation. REAL-type values are always printed with a decimal point, and only those constants which include a decimal point are actually stored as REAL values.

REAL numbers are stored in 8 consecutive memory bytes. The representation is based on the double-precision format of IEEE Draft Standard 754. Bit 7 of the first byte is the sign of the mantissa. Bits 0-6 of the first byte and bits 4-7 of the second byte form the exponent. The exponent is biased by 1024. The remaining 52 bits comprise the mantissa. The mantissa has an implied leading "1" bit.

INTERNAL REPRESENTATION OF REAL NUMBERS



|-- Mantissa Sign

The exponent covers the range 2.2*10 -308 (2 -1022 through 1.8*10 308 (2 1024) as powers of 2. Operations which result in values out of the representation range cause overflow or underflow errors (which may be handled automatically by the ON ERROR command).

The mantissa covers the range from 1.0 through 1.99999999999999 in steps of 2 -31. This means that REAL numbers can represent values on the number line about .0000000005 apart. Operations which cause results between the directly representable points are rounded to the nearest exactly representable number.

Floating point arithmetic is inherently inexact, thus a sequence of operations can produce a cumulative error. Proper rounding (as implemented in BASIC09) reduces this effect but can not eliminate it. Programmers using comparisons on REAL quantities should use caution with strict comparisons (i.e., =, or <>), since the exact desired value may not occur during program execution.

The STRING Data Type

A STRING is a variable length (0 or more) sequence of characters A STRING of zero characters is called an empty STRING. A variable may be defined as a STRING either explicitly (e.g., DIM title:STRING) or implicitly by appending a dollar-sign character to the variable name (e.g., title$:= "My First Program.").

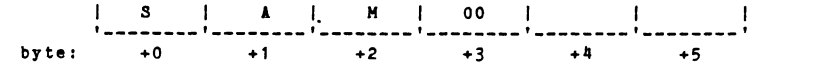
The default maximum length allocated to each string is 32 characters, but each string may be dimensioned less (e.g., DIM A:STRING [4]) for memory savings or more (e.g., DIM long:STRING [2880]) to allow long strings.

NOTE: Strings are inherently variable-length entities, and dimensioning the storage for a string only defines the maximum length string which can be stored there.

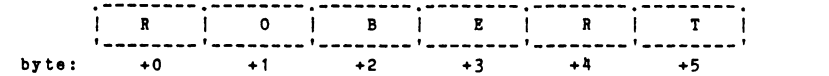
When a STRING value is assigned to a STRING variable, the bytes composing the string are copied into the variable storage byte-by-byte. The beginning of a string is always character number one, and this is NOT affected by the BASE0 or BASE1 statements. Operations which result in strings too long to fit in the dimensioned storage truncate the string on the right and no error is generated.

Normally the internal representation of the string is hidden from the user. A string is stored in a fixed-size storage area and is represented by a sequence of bytes terminated by the value zero or by the maximum length allocated to the STRING variable. Any remaining "unused" storage after the zero byte allows the stored string to expand and contract during execution.

The example below shows the internal storage of a variable dimensioned as STRING[6] and assigned a value of "SAM". Notice the byte at +3 contains the zero string terminator, and the two following bytes are not used.



If the value "ROBERT" is assigned to the variable the zero byte terminator is not needed because the STRING fills the storage exactly:



The BOOLEAN Data Type

A BOOLEAN data type can have only two values: TRUE or FALSE. They are stored as single byte values, but they may not be used for numeric computation. A variable may be typed BOOLEAN (e.g., DIM done_flag:**BOOLEAN**). BOOLEAN values print out as the character strings: "TRUE" and "FALSE".

BOOLEAN values result from comparing two compatible types and are appropriate for logical flags and expressions. For example, result:=a **AND** b **AND** c.

Do not confuse BOOLEAN operations **AND**, **OR**, **XOR** and **NOT** with the logical functions **LAND**, **LOR**, **LXOR** and **LNOT**. Logical functions use integer values to produce results on a bit-by-bit basis.

Attempting to store a non-BOOLEAN value to a BOOLEAN variable (or the reverse) will cause a binding error or an error on the second compiler pass when leaving edit mode.

AUTOMATIC TYPE CONVERSION

Expressions that mix numeric data types (BYTE, INTEGER or REAL) are automatically and temporarily converted to the largest type necessary to retain accuracy. In addition, certain BASIC09 functions also perform automatic type conversions as necessary. Thus, numeric quantities of mixed types may be used in most cases.

Type-mismatch errors happen when an expression includes types that can not legally be mixed. These errors are reported by the second compiler pass which automatically occurs when you leave EDIT mode. Type conversions can take time so it is advisable to use expressions containing all values of a single type wherever possible.

CONSTANTS

Constants are frequently used in program statements and in expressions to assign values to variables. BASIC09 has rules that allow you to specify constants that correspond to the five basic data types.

Numeric Constants

Numeric constants can be either type REAL or type INTEGER. If a number constant includes a decimal point or uses the "E format" exponential form, it forces BASIC09 to store the number in REAL format. This is true even if the value could be represented by an INTEGER or BYTE data type: for example 12.0.

Therefore, if you specifically want to use a REAL constant, include a decimal point. This is sometimes done if all other values in an expression are of type REAL so BASIC09 does not have to do a time-consuming type conversion at run-time.

Numbers that do not have a decimal point but are too large to be represented as integers are also stored in REAL format.

```
EXAMPLES:  1.0              9.8433218
           -.01             -999.000099
           100000000        5655.34532
           1.95E+12         -99999.9E-33
```

Numbers that do not have a decimal point and are in the range of -2,147,483,648 to +2,147,483,647 are treated as INTEGER numbers. BASIC09 will also accept integer constants in hexadecimal in the range 0 to $FFFFFFFF. Hex numbers must have a leading dollar sign.

```
EXAMPLES:   12       -3000    64000
           $20       $FFFE    $0
           0         -12      -32768
```

Boolean Constants

The two legal boolean constants are "TRUE" and "FALSE".

```
EXAMPLE:  DIM flag, state: BOOLEAN
          flag := TRUE
          state := FALSE
```

String Constants

String constants consist of a sequence of any characters enclosed in quotation marks. The binary value of each character byte can be 1 to 255. Quotation marks can be included in the string by using two quotation marks in a row to represent one quotation mark.

The null string "" is important because it represents a string having no characters. It is analogous to the numeric zero.

```
EXAMPLES:  "BASIC09 is a new microcomputer language"
           "AABBCCDD"
           ""   (a null string)
           "An ""older woman"" is wiser"
```

VARIABLES

Each BASIC09 variable is "local" to the procedure where it is defined. Local means that it is only known to the program statements within that procedure. You can use the same variable name in several procedures and the variables will be completely independent. If you specifically want other procedures to be able to share a variable, you must use the RUN and PARAM statements to pass the variable when a procedure is calling another procedure.

Storage for variables is allocated from the BASIC09 workspace when the procedure is called. It is not possible to force a variable to occupy a particular absolute address in memory. When the procedure is exited, variable storage is given back and values stored in it are lost. Procedures can call themselves (this is referred to as **recursion**) which causes another separate storage space for variables to be allocated.

WARNING!! BASIC09 DOES NOT AUTOMATICALLY INITIALIZE VARIABLES. WHEN A PROCEDURE IS RUN ALL VARIABLES, ARRAYS AND STRUCTURES WILL HAVE RANDOM VALUES. YOUR PROGRAM MUST ASSIGN ANY INITIAL VALUE IF NEEDED.

Parameter Variables

Procedures may pass variables to other procedures. When this occurs, the variables passed to the called procedure are referred to as "parameters". Parameters may be passed in two ways;

by reference This allows values to be returned from the called procedure to calling procedure variables.

by value This protects the values in the calling procedure so that they may not be changed by the procedure which is called).

Parameters are usually passed "by reference". This is done by enclosing the names of the variables to be sent to the called procedure in parentheses as part of the RUN statement. The storage address of each parameter variable is evaluated and sent to the called procedure which then associates those addresses with names in a local PARAM statement.

The called procedure uses this storage as if it had been created locally (although it may have a new name) and can change the values stored there. Parameters passed by reference allow called procedures to return values to their callers.

Parameters may be passed "by value" by writing the value to be passed as an expression which is evaluated at the time of the call. Useful expression-generators that do not alter values are +0 for numbers or +"" for strings. For example:

```
RUN inverse(x)              passes "x" by reference
RUN inverse(x+0)            passes "x" by value
RUN translate(word$)        passes "word$" by reference
RUN translate(word$+"")     passes "word$" by value
```

When parameters are passed by value, a temporary variable is created when the expression is evaluated. The result is placed in temporary storage. The address of this temporary storage is sent to the called procedure. Therefore, the value actually given to the called procedure is a copy of the result, and the called procedure can not accidentally (or otherwise) change the variable(s) in the calling program.

Notice that expressions containing numeric constants will be either of type INTEGER or of type REAL; there is no type BYTE constant. Thus, BYTE-type VARIABLES may be sent to a procedure as parameters but expressions will be of types INTEGER or REAL. For example, a RUN statement may evaluate an INTEGER as a parameter and send it to the called procedure. If the called procedure is expecting a BYTE-type variable, it will use only the high-order byte of the (four-byte) INTEGER (which, if the value was intended to be in BYTE-range, will probably be zero!).

ARRAYS

The DIM statement can be used to create arrays of from 1 to 3 dimensions. A one-dimensional array is often called a "vector". A 2 dimensional array is called a "table". A 3 dimensional array is called a "matrix". The sizes of each dimension are defined when the array is typed (e.g., DIM plot(24,80):BYTE) by including the number of elements in each dimension.

Therefore, a table dimensioned (24,80) has 24 rows (1-24) of 80 columns (1 - 80) when accessed in the default (BASE 1) mode. Programmers may elect to access the elements of an array starting at zero (BASE 0), in which case there are still 24 rows (now 0-23) and 80 columns (now 0-79). Arrays may be composed of basic data types, complex data types or other arrays.

COMPLEX DATA TYPES

The TYPE statement can be used to define a new data type as a "vector" (a one-dimensional array) of any basic or previously-defined types. For example:

```
TYPE employee_rec = name:STRING; number(2):INTEGER; malesex:BOOLEAN
```

This structure differs from an array in that the various elements may be of mixed types, and the elements are accessed by a field name instead of an array index. For example:

```
DIM employee_file(250): employee_rec
employee_file(1).name := "Tex"
employee_file(20).number(2) := 115
```

The complex structure gives the programmer the ability to store and manipulate related values that are of many types, to create "new" types in addition to the five atomic data types, or to create data structures of unusual "shape" or size. The position of the desired element in complex-type storage is known and defined at "compile time" and need not be calculated at "run time". Therefore, complex structure accesses may be slightly faster than array accesses.

The elements of a complex structure may be copied to another similar structure using a single assignment operator (i.e., ":="). An entire structure may be written to or read from mass storage as a single entity (e.g., PUT #2, employee_file).

Arrays or complex structures may be elements of subsequent complex structures or arrays.

end of chapter 8

EVALUATION OF EXPRESSIONS

Many BASIC09 statements evaluate expressions. The result of an evaluation is always a value of some basic type: REAL, INTEGER, STRING, or BOOLEAN. The expression itself may consist of values and operators. For example, the expression "5+5" results in an integer with a value of ten.

A "value" can be a constant value, a variable name or a function which "returns" the result as a value. An operator combines values (typically, those adjacent to the operator) and also returns a result.

In the course of evaluating an expression, each value is copied onto an "expression stack" where functions and operators take their input values and return results. If the expression is to be used in an assignment statement, only when the result of the entire expression has been found is the assignment made. This allows the variable which is being modified to be one of the values in the expression. The same principles apply for numeric, string, and boolean operators. These principles make assignment statements such as "X=X+1" legal in all cases even though it would not make sense in a mathematical context.

Any expression will evaluate to one of the five basic data types. This does not mean, however, that all the operators and operands in expressions have to be of an identical type. Often types are mixed in expressions because the RESULT of some operator or function has a different type than its operands. An example is the "less than" operator:

```
24 < 100
```

The "<" operator compares two numeric operands. The result of the comparison is of type BOOLEAN; in this case, the value TRUE.

BASIC09 allows intermixing of the three numeric types because it performs automatic type conversion of operands. If different types are used in an expression, the "result" will be the same type as the operand(s) having the largest representation. As a rule, any numeric type operand may be used in a expression that is expected to produce a result of type REAL. Expressions that must produce BYTE or INTEGER results must evaluate to a value that is small enough to fit the representation. BASIC09 has a complete set of functions that can perform compatible type conversion. Type-mismatch errors are reported by the second compiler pass when leaving Edit mode.

OPERATORS

Operators (except negation) cause some operation to be performed on two operands. This produces a result, which is generally the same type as the operands (except comparisons). The table below lists the operators available and the types they accept and produce.

Operator	Function	Operand Type	Result Type
-	Negation	NUMERIC	NUMERIC
or **	Exponentiation	NUMERIC (positive)	NUMERIC
*	Multiplication	NUMERIC	NUMERIC
/	Division	NUMERIC	NUMERIC
+	Addition	NUMERIC	NUMERIC
-	Subtraction	NUMERIC	NUMERIC
NOT	Logical Negation	BOOLEAN	BOOLEAN
AND	Logical AND	BOOLEAN	BOOLEAN
OR	Logical OR	BOOLEAN	BOOLEAN
XOR	Logical EXCLUSIVE OR	BOOLEAN	BOOLEAN
+	Concatenation	STRING	STRING
=	Equal to	ANY	BOOLEAN
<> or ><	Not equal to	ANY	BOOLEAN
<	Less than	NUMERIC, STRING*	BOOLEAN
<= or =<	Less than or Equal	NUMERIC, STRING*	BOOLEAN
>	Greater than	NUMERIC, STRING*	BOOLEAN
>= or =>	Greater than or Equal	NUMERIC, STRING*	BOOLEAN

* When comparing strings, the ASCII collating sequence is used, so that 0 < 1 < ... < 9 < A < B < ... < Z < a < b < ... < z

OPERATOR PRECEDENCE

Operators have "precedence" which means they are evaluated in a specific order. Parentheses can be used to override natural precedence, however, extraneous parentheses may be removed by the compiler. The legal operators are listed below, in precedence order from highest to lowest.

Highest Precedence.

```
NOT     -(negate)

        **

*       /

+       -

>       <       <>      =       >=      <=

AND

OR      XOR
```

Lowest precedence

Operators of equal precedence are shown on the same line, and are evaluated left to right in expressions. The only exception to this rule is exponentiation, which is evaluated right to left. Raising a negative number to a power is not legal in BASIC09.

In the examples below, BASIC09 expressions on the left will be evaluated as indicated on the right. Either form may be entered, but the simpler form on the left will always be generated by the decompiler.

BASIC09 Representation	Equivalent form
a:= b+c**2/d	a:= b+((c**2)/d)
a:= b>c AND d>e OR c=e	a:= ((b>c) AND (d>e)) OR (c=e)
a:= (b+c+d)/e	a:= ((b+c)+d)/e
a:= b**c**d/e	a:= (b**(c**d))/e
a:= -(b)**2	a:= (-b)**2
a=b=c	a:= (b=c) (return BOOLEAN value)

FUNCTIONS

Functions take one or more arguments enclosed in parentheses, perform some operation and return a value. They may be used as operands in expressions. Functions expect that the arguments passed to them will be expressions, constants or variables of a certain type and will return a result of a certain type. Giving a function an argument of an incompatible type will result in an error.

In the descriptions of functions that follow, the following notation is used to describe the type required for the parameter expressions:

<num>	means any numeric-result expression
<str>	means any string-result expression
<int>	means any integer-result expression

The functions below return REAL results. Accuracy of transcendental functions is 8+ decimal digits. Angles can be either degrees or radians (see DEG/RAD statement descriptions).

NOTE: Transcendental functions will take a long time to return a value if passed a extremely large value (i.e., SIN(100000000)).

NAME	FUNCTION
SIN(<num>)	trigonometric sine of <num> RESULT: -1 <= SIN(<num>) <= 1
COS(<num>)	trigonometric cosine of <num> RESULT: -1 <= COS(<num>) <= 1
TAN(<num>)	trigonometric tangent of <num>
ASN(<num>)	trigonometric arcsine of <num> RESULT: -PI/2 <= ASN(<num>) <= PI/2
ACS(<num>)	trigonometric arcosine of <num> RESULT: 0 <= ACS(<num>) <= PI
ATN(<num>)	trigonometric arctangent of <num> RESULT: -PI/2 <= ATN(<num>) <= PI/2
LOG(<num>)	natural logarithm (base e) of <num>, which must be positive
LOG10(<num>)	logarithm (base 10) of <num>, which must be positive
SQR(<num>)	square root of <num>, which must be positive
SQRT(<num>)	square root of <num>; same as SQR
EXP(<num>)	e (2.71828183) raised to the power <num>, which must be a positive number.
FLOAT(<num>)	<num> converted to type REAL (from BYTE or INTEGER)
INT(<num>)	truncates all digits to the right of the decimal point of REAL <num> Examples: INT(0.21) = 0. INT(-2.5) = -2.

NAME	FUNCTION
PI	the constant 3.141592653589793
RND(<num>)	If <num>=0 returns random x, 0 <= x < 1 if <num>>0 returns random x, 0 <= x < <num> if <num><0 use ABS(<num>) as new random number seed

The following functions return results of type INTEGER or BYTE:

NAME	FUNCTION
FIX(<num>)	round REAL <num> and convert to type INTEGER
MOD(<num1>,<num2>)	modulus (remainder) function. MOD(<num1>,<num2>) = the remainder of <num1> divided by <num2>. If <num1> is negative, the result will be negative. Examples: MOD(9,5) = 4 MOD(-11,3) = -2
ADDR(<name>)	absolute memory address of variable, array, or structure named <name>.
SIZE(<name>)	storage size in bytes of variable, array, or structure named <name>.
ERR	error code of most recent error, automatically resets to zero when referenced
PEEK(<int>)	value of byte at memory address <int> WARNING: the specified memory address must be within the limits of accessable memory space. PEEK(-1) will give a bus error on most systems. This will cause BASIC09 to abort.
POS	current character position of PRINT buffer
ASC(<str>)	numeric value of first character of <str>
LEN(<str>)	length of string <str>

NAME	FUNCTION
SUBSTR(<str1>,<str2>)	substring search: returns starting position of first occurrence of <str1> in <str2>, or 0 if not found.
INKEY(#<num>)	returns the number of characters in data buffer
FILSIZ(#<num>)	returns size of a file

The following functions can return ANY numeric type, depending on the type of the input parameter:

NAME	FUNCTION
ABS(<num>)	absolute value of <num>
SGN(<num>)	signum of <num>: -1 if <num> < 0, 0 if <num> = 0, or 1 if <num> > 0
SQ(<num>)	square <num>
VAL(<str>)	convert type STRING to type NUMERIC

The following functions perform bit-by-bit logical operations on integer or byte data types and return integer results. They should NOT be confused with the BOOLEAN-type operators.

NAME	FUNCTION
LAND(<num>,<num>)	Logical AND
LOR(<num>,<num>)	Logical OR
LXOR(<num>,<num>)	Logical EXCLUSIVE OR
LNOT(<num>)	Logical NOT

These functions return a result of type STRING:

NAME	FUNCTION
CHR$(<int>)	ASCII character equivalent of <int> <int> must be within range of 0-127
DATE$	date and time, format: "yy/mm/dd hh:mm:ss"
LEFT$(<str>,<int>)	leftmost <int> characters of <str>
RIGHT$(<str>,<int>)	rightmost <int> characters of <str>
MID$(<str>,<int1>,<int2>	middle <int2> characters of <str> starting at character position <int1>
STR$(<num>)	converts numeric type <num> to displayable characters of type STRING representing the number converted.
TRIM$(<str>)	<str> with trailing spaces removed

The following functions return BOOLEAN values:

NAME	FUNCTION
TRUE	always returns TRUE
FALSE	always returns FALSE
EOF(#<num>)	End-of-file test on disk file path <num>, returns TRUE if end-of-file condition exists.

end of chapter 9

PROGRAM STRUCTURE

A BASIC09 program can be written as a single procedure, or it may be divided into a number of smaller procedures, each of which is designed to perform a specific function.

Single procedure programs may be useful when the program is relatively small. However, large complex programs are generally much easier to develop, test and maintain when the program is divided into several procedures. To accomplish this, the programmer should create a main routine which will call other BASIC09 procedures to perform specific functions as subroutines. These BASIC09 procedures may in turn call other BASIC09 procedures in the same manner. These techniques reflect sound structured programming practice.

A procedure consists of any number of program statement lines. Each line can have an optional line number. More than one program statement can be put on the same line if separated by "\" characters. For example, the following statements are equivalent:

```
GOSUB 550 \ PRINT X,Y \ RETURN
```

```
GOSUB 550
PRINT X,Y
RETURN
```

While the above statements are functionally equivalent, the second is generally considered preferable. The first method runs no faster and tends to hide the structure of the program.

The number of characters on a given line is dependent on the content of the line. In general, lines should be limited to 128 characters or less, to avoid the generation of errors when BASIC09 decompiles the I-Code for listing purposes or at run time.

Loop nesting is limited to 39 levels. Nested procedure and subroutine calls are only limited by stack space.

Program readability is improved if all variables are declared with DIM statements at the beginning of the procedure, but this is not mandatory.

Line numbers are optional. They can be any integer number in the range of 1 to 32767. Line numbers should only be used where absolutely necessary (such as with GOSUB). They make programs harder to understand, use additional memory space and increase compile time considerably. Line numbers are local to procedures, i.e., the same line number can be used in different procedures without conflict.

The program can be terminated with END or STOP statements, which are also optional.

ASSIGNMENT STATEMENTS

Assignment statements are used for computation or initialization of variables. The two assignment statements available with BASIC09 are LET and POKE.

LET Statement

SYNTAX: [LET] <var> := <expr>
[LET] <var> = <expr>
[LET] <struct> := <struct>
[LET] <struct> = <struct>

FUNCTION: LET evaluates an expression and stores the result in <var> which may be a simple variable or data structure element. The result of the expression (<expr>) must be of the same or compatible type as <var>.

BASIC09 will accept either "=" or ":=" as an assignment operator, however, the second form (:=) is preferred because it distinguishes the assignment operation from a comparison (the test for equality). The ":=" operator is the same as is used in PASCAL.

Another use of the assignment statement is to copy the entire value of an array or complex data structure to another array or complex data structure. The data structures do not have to have the same type or "shape". The only restriction is that the size of the destination structure be the same or larger than the source structure.

This type of assignment can be used to perform unusual type conversions. For example, a string variable of 80 characters can be copied to a one-dimensional array of 80 bytes.

EXAMPLES:
```
A := 0.1

value := temp/sin(x)

DIM array1(100), array2(100)
array1 := array2

LET AUTHOR$ := FIRST_NAME$ + LAST_NAME$

DIM truth,lie:BOOLEAN
lie := 100 < 1
truth := NOT lie

count = total-adjustment

matrix(2).coefficient(n+2) := matrix(1).coefficient(n)
```

POKE Statement

SYNTAX: POKE <integer expr> , <byte expr>

FUNCTION: POKE allows a program to store data at a specific memory address. The first expression is used as the absolute address to store the type BYTE result of the second expression. POKE can alter any memory address, so care should be taken when using it.

WARNING: Using POKE with an invalid address will cause BASIC09 to abort.

EXAMPLES:
```
POKE ADDR(buffer)+5,ASC("A")

POKE 1200,14

POKE $1C00,$FF

POKE pointer,PEEK(pointer+1)

(* same as alphabet$ := "ABCDEFGHIJKLMNOPQRSTUVWXYZ" *)
FOR i=0 to 25
  POKE ADDR(alphabet$)+i,$40+i
NEXT i
POKE ADDR(alphabet$)+26,$FF
```

CONTROL STATEMENTS

Control statements affect the sequential execution of program statements. They are used to construct loops or make decisions that alter program flow. BASIC09 provides a selection of loop statements that allow any kind of loop to be constructed using sound structured programming style. The control statements are IF..THEN..ELSE, FOR..NEXT, WHILE..DO, REPEAT..UNTIL, LOOP..ENDLOOP, GOTO, GOSUB..RETURN, ON GOTO..RETURN, ON GOSUB..RETURN and ON ERROR GOTO.

IF..THEN..ELSE Statement

SYNTAX:
```
IF <bool expr> THEN <statements>
[ ELSE <statements> ]
ENDIF

IF <bool expr> THEN <line #>
[ ELSE <statements> ]
ENDIF
```

FUNCTION: The IF structure evaluates an expression to a BOOLEAN value. If the result is TRUE the statement(s) immediately following the THEN are executed. If an ELSE clause exists, statements between the ELSE and ENDIF are skipped. IF the expression evaluated to FALSE control is transferred to the first statement following the ELSE (if present) or to the statement immediately following the ENDIF.

A special form of the IF statement causes execution to be transferred to the statement having a line number specified if the result of the expression is TRUE. The line number follows immediately after THEN. This is an implied GOTO statement. It should be used with extreme caution (as all GOTO statements should).

EXAMPLES:
```
IF a < b THEN
  PRINT "a is less than b"
  PRINT "a:";a;" b:";b
ENDIF

IF a < b THEN
  PRINT "a is less than b"
ELSE
  IF a=b THEN
    PRINT "a equals b"
  ELSE
    PRINT "a is greater than b"
  ENDIF
ENDIF

IF payment < balance THEN 400
```

FOR..NEXT Statement

SYNTAX: FOR <var> = <expr> TO <expr> [STEP <expr>]
NEXT <var>

FUNCTION: The FOR..NEXT statement creates a loop that usually executes a given number of times while automatically increasing or decreasing a specified counter variable.

The first expression is evaluated and the result stored in <var>. <var> must be a simple integer or real variable. The second expression is evaluated and stored in a temporary variable.

If the STEP clause is used, its expression is evaluated and used as the loop increment. If the increment is negative, the loop will count DOWN.

The "body" of the loop (i.e. statements between the "FOR" and "NEXT") is executed until the next variable(a counter) is larger than the terminating expression value. For negative STEP values, the loop will execute until the loop counter is less than the termination value. If the initial value of <var> is beyond the terminating value, the body of the loop is never executed.

It is legal to jump out of FOR..NEXT loops.

CAVEAT: When using REAL control and STEP expressions there is a possibility of not completing the number of loops logically indicated. For example, the following loop would seem to complete "x" number of loops. Due to the rounding nature of REAL numbers, it may only complete "x - .1" loops:

```
FOR a = 1/x TO 1 STEP 1/x
  .
  .
next a
```

To make sure that the loop is executed the correct number of times, use INTEGER values for all expressions.

EXAMPLES:

```
FOR var = min-1 TO min+max STEP increment-adjustment
  PRINT var
NEXT var


FOR x = 1000 TO 1 STEP -1
  PRINT x
NEXT x
```

WHILE..DO Statement

SYNTAX:
```
WHILE <bool expr> DO
  <statements>
ENDWHILE
```

FUNCTION: This is a loop construct which tests its control expression at the "top" of the loop. Statements within the loop are executed as long as <bool expr> is TRUE. The body of the loop will not be executed if the boolean expression evaluates to FALSE when first executed.

EXAMPLES:
```
WHILE a<b DO      is equivalent to      100 IF a>=b THEN 500
  PRINT a                                   PRINT a
  a := a+1                                  a := a+1
ENDWHILE                                    GOTO 100
                                        500 REM
```

```
DIM yes:BOOLEAN
yes=TRUE
WHILE yes DO
  PRINT "yes!  ";
  yes := POS<50
ENDWHILE
```

```
REM reverse the letters in word$
backward$ := ""
INPUT word$
WHILE LEN(word$) > 0 DO
  backward$ := backward$ + RIGHT$(word$,1)
  word$ := LEFT$(word$,LEN(word$)-1)
ENDWHILE
word$ := backward$
PRINT word$
```

REPEAT..UNTIL Statement

SYNTAX: REPEAT
<statements>
UNTIL <bool expr>

FUNCTION: This is a loop that tests its control expression at the bottom of the loop. The statement(s) within the loop are executed until the result of <bool expr> is TRUE. The body of the loop is always executed at least one time.

EXAMPLES:

```
x = 0                    is the same as          x=0
REPEAT                                       100 PRINT x
  PRINT x                                        x=x+1
  x=x+1                                          IF X <= 10 THEN 100
UNTIL x>10
```

```
(* compute factorial: n! *)
temp := 1.
INPUT "Factorial of what number? ",n
REPEAT
  temp := temp * n
  n := n-1
UNTIL n <= 1.0
PRINT "The factorial is "; temp
```

LOOP..ENDLOOP / EXITIF..ENDEXIT Statements

```
SYNTAX:  LOOP
         <statements>
         ENDLOOP

         EXITIF <bool expr> THEN <statements>
         ENDEXIT
```

FUNCTION: The LOOP..ENDLOOP and the EXITIF..ENDEXIT statements are inherently related. They can be used to construct loops with tests located any place in the body of the loop. The LOOP and ENDLOOP statements define the body of the loop. EXITIF clauses can be inserted anywhere inside the loop to leave the loop if the result of its test is true.

NOTE: If there is no "exit" clause you will create an endless loop.

The EXITIF clause evaluates an expression to a boolean result. If the result is TRUE, the statements between the THEN and the ENDEXIT are executed and then control is transferred to the first statement following ENDLOOP. Otherwise, the statement following ENDEXIT is executed. This exit clause is often used to perform some specific function upon termination of the loop which depends on where the loop terminated.

EXITIF statements are almost always used when LOOP..ENDLOOP is used, but they can also be useful in ANY type of loop construct.

EXAMPLES:

```
LOOP                          is equivalent to   100 count=count+1
count=count+1                                        IF count <= 100 THEN
EXITIF count >100 THEN                                 PRINT count
  done = TRUE                                          GOTO 100
ENDEXIT                                              ELSE
  PRINT count                                          done = TRUE
ENDLOOP                                              ENDIF
                                                     REM out of loop
```

```
INPUT x,y
LOOP
  PRINT
EXITIF x < 0 THEN
  PRINT "x became zero first"
ENDEXIT
  x := x-1
EXITIF y < 0 THEN PRINT "y became zero first"
ENDEXIT
  y := y-1
ENDLOOP
```

GOTO Statement

SYNTAX: GOTO <line #>

FUNCTION: The GOTO unconditionally transfers execution flow to the line having the specified number. Note that the line number is a constant, not an expression or a variable.

EXAMPLE: GOTO 1000

GOSUB..RETURN Statements

SYNTAX:

```
GOSUB <line #>
  .
  .
<line#> <statements>
RETURN
```

FUNCTION: The GOSUB statement transfers program execution to a subroutine starting at the specified line number. The subroutine is executed until a RETURN statement is encountered which causes execution to resume at the statement following the calling GOSUB. Subroutines may be "nested" to any depth.

EXAMPLE:

```
FOR n := 1 to 10
  x := SIN(n)
  GOSUB 100
NEXT n
FOR m := 1 TO 10
  x := COS(m)
  GOSUB 100
NEXT m
STOP

100 x := x/2
    PRINT x
    RETURN
```

ON GOTO Statement / ON GOSUB Statement

SYNTAX: ON <integer expr> GOTO <line #> [,<line #>]
ON <integer expr> GOSUB <line #> [,<line #>]

FUNCTION: These statements evaluate an integer expression and use the result to select a corresponding line number from an ordered list. Control is then unconditionally transferred to that line number in ON GOTO statements or as a subroutine in ON GOSUB statements.

These statements are similar to CASE statements in other languages.

Each <line #> in the ordered list is given a value (beginning with 1, 2, 3, ...). The <integer expr> must evaluate to a positive INTEGER result having a value of between 1 and N; N being the highest line number in the list. N is limited by input line length and the number of digits in each line number. The best case limit for N is 60.

NOTE: If the expression has a result that does not correspond with the ordered list, no GOSUB statement is selected and the next sequential statement is executed.

EXAMPLE:

```
(* spell out the digits 0 to 7 *)
DIM digit:INTEGER
A$="one digit only, please"
INPUT "type in a digit"; digit
ON digit+1 GOSUB  10,11,12,13,14,15,16,17
PRINT A$
STOP

(* names of digits *)
10 A$ := "ZERO"
   RETURN
11 A$ := "ONE"
   RETURN
12 A$ := "TWO"
   RETURN
13 A$ := "THREE"
   RETURN
14 A$ := "FOUR"
   RETURN
15 A$ := "FIVE"
   RETURN
16 A$ := "SIX"
   RETURN
17 A$ := "SEVEN"
   RETURN
```

ON ERROR GOTO Statement

SYNTAX: ON ERROR [GOTO <line #>]

FUNCTION: This statement sets a "trap" that transfers control to the line number given when a non-fatal run-time error occurs. If no ON ERROR GOTO has been executed in a procedure before an error occurs, the procedure will stop and enter DEBUG mode. The error trap can be turned off by executing ON ERROR without a GOTO.

This statement is often used in conjunction with the ERR function which returns the specific error code, and the ERROR statement which artificially generates "errors".

NOTE: The ERR function automatically resets to zero any time it is called.

EXAMPLE:

```
(* List a file *)

 DIM path,errnum: INTEGER, name: STRING[45]
 DIM line: STRING[80]
 ON ERROR GOTO 10
 INPUT "File name? "; name
 OPEN #path,name:READ
 LOOP
   READ #path, line
   PRINT line
 ENDLOOP

10 errnum=ERR
   IF errnum := 211 THEN
   (* end-of-file *)
     PRINT "Listing complete."
     CLOSE #path
     END
   ELSE
   (* other errors *)
     PRINT "Error number "; errnum
     END
   ENDIF
```

EXECUTION Statements

Execution statements are used to run procedures, stop execution of procedures, create Shells or affect the current execution of the procedure.

RUN Statement

SYNTAX: RUN <proc name> [(*<param> {,<param>})]

FUNCTION: This statement calls a procedure by name. When that procedure ends, control will pass to the next statement after the RUN statement. It is most often used to call a procedure inside the workspace, but it can also be used to call a previously compiled procedure or a 68000 machine language procedure outside the workspace. The name can be optionally taken from a string variable.

Parameter Passing

The RUN statement can include a list of parameters enclosed in parentheses to be passed to the called procedure. The called procedure must have PARAM statements of the same size and order to match the parameters passed to it by the calling procedure.

The parameters can be variables or constants, or the names of entire arrays or data structures. They can be of any type, EXCEPT variables of type BYTE. However, BYTE arrays are allowed.

If a parameter is a constant or expression, it is passed "by value". A parameter passed by value is evaluated and placed in a temporary storage location and the address of the temporary storage is passed to the called procedure. Parameters passed by value can be changed by the receiving procedure, but the changes are not reflected in the calling procedure.

If the parameter is the name of a variable, array, or data structure it is passed by "reference". When passed by referance, the address of that storage is sent to the called procedure and thus the value in that storage may be changed by the receiving procedure. These changes ARE reflected in the calling procedure.

Calling External Procedures

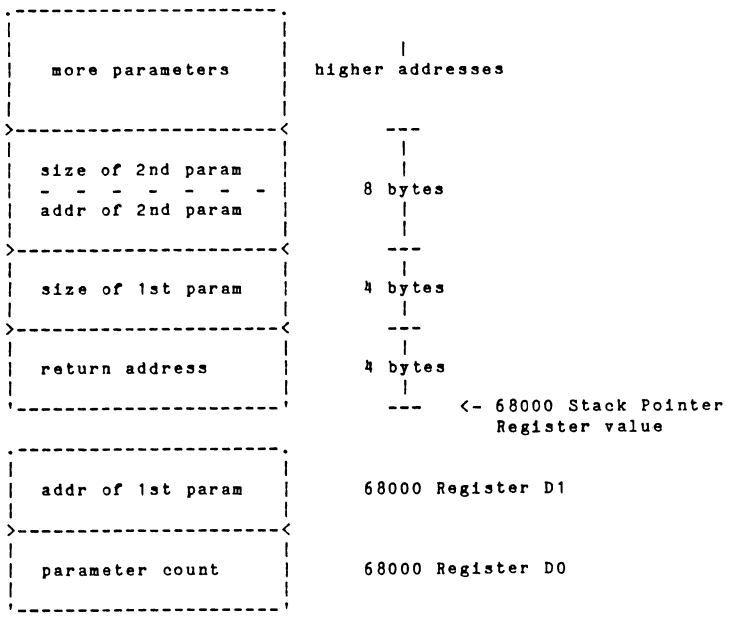
If the procedure named by the RUN statement can not be found in the workspace, BASIC09 will check to see if it was loaded by OS-9 outside the workspace. If it is not found there, BASIC09 will try to find a disk file having the same name in the current execution directory, load it and run it.

In either case, BASIC09 checks to see if the called procedure is a BASIC09 I-code module or a 68000 machine language module and executes it accordingly. If it is a 68000 machine language module, BASIC09 executes a JSR instruction to its entry point and the module is executed as 68000 native code. The machine language routine can return to the original calling procedure by executing an RTS instruction. The diagram on the next page shows what the stack frame passed to machine-language subroutines looks like.

Machine language modules return error status by setting the carry bit of the MPU condition codes register and by setting the low order word of register D1 to the appropriate error code. For an example of a machine language subroutine ("SYSCALL"), see Appendix A.

After an external procedure has been called but is no longer needed, the KILL statement should be used to get rid of it so its memory space can be used for other purposes.

STACK FRAME PASSED TO MACHINE LANGUAGE PROCEDURES



EXAMPLE:

```
PROCEDURE trig_table
num1 := 0 \ num2 := 0
REPEAT
  RUN display(num1,SIN(num1))
  RUN display(num2,COS(num2))
  PRINT
UNTIL num1 > 1
END

PROCEDURE display
PARAM passed,funcval
PRINT passed;":";funcval,
passed := passed + 0.1
END
```

KILL Statement

SYNTAX: KILL <proc name>

FUNCTION: This statement is used to "unlink" an external procedure, possibly returning system memory, and remove it from BASIC09's procedure directory. If the procedure is inside the workspace, nothing happens and no error is generated. KILL can be used along with auto-loading PACKed procedures as an alternative to CHAIN when program overlay is desired.

The procedure name may be optionally called from a string variable.

WARNINGS:
1. It can be fatal to OS-9 to KILL a procedure that is still "active".
2. When the KILL statement is used with a RUN statement, both statements MUST use the same string variable which contains the name of the procedure. See first example below:

EXAMPLES:

```
LET procname$="average"
RUN procname$
KILL procname$

INPUT "Which test do you want to run? ",test$
RUN test$
KILL test$
```

CHAIN Statement

SYNTAX: CHAIN <SHELL command line>

FUNCTION: The CHAIN statement performs an OS-9 "chain" operation on the SHELL, passing the specified command line as an argument. CHAIN causes BASIC09 to be exited, unlinked, and its memory to be returned to OS-9. The command line should evaluate to the name of an executable module (such as BASIC09), passing parameters if appropriate.

CHAIN can begin execution of any module, not just BASIC09. It executes the module indirectly through the SHELL in order to take advantage of SHELL's parameter processing, which has the side-effect of leaving an extra "incarnation" of the SHELL active. Programs that repeatedly chain to each other will eventually find all of memory filled with waiting SHELLs. This can be prevented by using the "ex" option of SHELL.

Consult the "OS-9/68000 OPERATING SYSTEM USER'S MANUAL" for more details on the capabilities of the SHELL.

NOTE: If it is necessary to pass an open path to another program, using the CHAIN command, the "ex" option of SHELL must be used.

Examples:
```
CHAIN "ex BASIC09 menu"

CHAIN "BASIC09 #10k sort (""datafile"",""tempfile"")"

CHAIN "DIR /D0"

CHAIN "Dir; Echo " Copying Directory "; ex basic09 copydir"
```

SHELL Statement

SYNTAX: SHELL [<SHELL command line>]

FUNCTION: SHELL allows BASIC09 programs to run any OS-9 command or program. SHELL gives access to virtually any OS-9 function including multiprogramming, utility commands, terminal and I/O control.

Consult the "OS-9/68000 OPERATING SYSTEM USER'S MANUAL" for a detailed discussion of OS-9 standard commands.

The SHELL statement requests OS-9 to create a new process, initially executing the "shell" which is the OS-9 command interpreter. The shell can then call any program in the system. The command line is evaluated and passed to the shell to be executed as a command line. If no command line is specified, BASIC09 is temporarily suspended and the shell process displays prompts and accepts commands in its normal manner. When the shell process terminates, BASIC09 becomes active again and resumes execution at the statement following the SHELL statement.

EXAMPLES:

`SHELL "copy file1 file2"`	sequential execution
`SHELL "copy file1 file2&"`	concurrent execution
`SHELL "edt document"`	calling text editor
`SHELL "asm source o=obj ! spl &"`	concurrent assembly
`SHELL ""`	transfer control to SHELL

END Statement

SYNTAX: END [<output list>]

FUNCTION: END ends execution of the procedure and returns to the calling procedure or to BASIC09 command mode if it was the highest level procedure. If an output list is given, END prints the list to standard output (the same as a PRINT statement).

END is an executable statement and can be used several times in the same procedure. END is optional; it is not required at the "bottom" of a procedure.

EXAMPLES:
```
END
END "I have finished execution"
```

STOP Statement

SYNTAX: STOP [<output list>]

FUNCTION: STOP immediately terminates execution of all procedures and returns to the Command Mode. If an output list is given, it also executes like a PRINT statement.

BYE Statement

SYNTAX: BYE

FUNCTION: BYE ends execution of the procedure and terminates BASIC09. Any open files are closed, and any unsaved procedures or data in the workspace will be lost. BYE is especially useful for creating packed programs and/or programs to be called from OS-9 procedure files.

WARNING: BYE CAUSES BASIC09 TO ABORT. IT SHOULD ONLY BE USED IF THE PROGRAM HAS BEEN SAVED BEFORE IT IS TESTED!

DIGITS Statement

SYNTAX: DIGITS [<int expr>]

FUNCTION: DIGITS controls the precision of real numbers displayed by a PRINT statement. If no <int expr> is specified, DIGITS will return the current precision.

DIGITS also controls the precision of transcendental functions. If the result of the expression is not between 1 and 15, the result is brought into that range with no error.

EXAMPLE:
```
PROCEDURE DIGITDEMO
DIM a,b : REAL
DIGITS 2
a := 9.2
b := 0.11
PRINT a,b,a*b

RUN DIGITDEMO
9.2             .11             1.
```

ERROR Statement

SYNTAX: ERROR(<integer expr>)

FUNCTION: ERROR generates an error having the error code specified by the result of evaluation of the expression. ERROR is often used for testing error routines. Also see the ON ERROR description.

PAUSE Statement

SYNTAX: PAUSE [<output list>]

FUNCTION: PAUSE suspends execution of the procedure and causes BASIC09 to enter Debug Mode. If an output list is given, it also executes as a PRINT statement. When a PAUSE is encountered the following will be displayed:

<output list> BREAK IN PROCEDURE <procedure name>

The Debug Mode "CONT" command can be used to resume procedure execution at the following statement.

EXAMPLES:
```
PAUSE

PAUSE "now outside main loop"
```

CHD and CHX Statements

SYNTAX: CHD <pathlist>
CHX <pathlist>

FUNCTION: The CHD and CHX statements change the current default Data or Execution directories, respectively. The pathlist must refer to a file which has the DIR attribute. For more information on the OS-9 directory structure, consult the "OS-9/68000 OPERATING SYSTEM USER'S MANUAL".

DEG and RAD Statements

SYNTAX: DEG
RAD

FUNCTION: These statements set the procedure's state flag to assume angles stated in degrees or radians in SIN, COS, TAN, ACS, ASN and ATN functions. This flag applies only to the currently active procedure. The default state is radians.

EXAMPLE:
```
DIM a : REAL
DEG
INPUT "enter degree of angle" a
PRINT "The sin of "; a; "degrees is "; SIN (a)
END
```

BASE 0 and BASE 1 Statements

SYNTAX: BASE 0
BASE 1

FUNCTION: These statements indicate whether a particular procedure's lowest array or data structure index is zero or one. The default is one. These statements do not affect the string operations where the beginning character of a string is always index one.

TRON and TROFF Statements

SYNTAX: TRON
TROFF

FUNCTION: These statements turn the trace mode on or off and are useful for debugging. When trace mode is turned on, each statement is decompiled and printed before execution. The result of each expression evaluation is printed as it occurs.

COMMENT STATEMENTS

REM / (*

SYNTAX: REM <chars>
(* <chars> [*)]

FUNCTION: These statements are used to put comments in programs. The second form of the statement is for compatibility with PASCAL programs. Comments are retained in the I-code but are removed by the PACK compile command. The "!" character can be typed in place of the keyword REM when editing programs. The compiler trims away extra spaces following REM to conserve memory space.

EXAMPLES:
```
REM this is a comment

(* This is also a comment *)

(* This is another kind of comment
```

DECLARATION STATEMENTS

The DIM, PARAM, and TYPE statements are called **declaration statements** because they are used to define and/or declare variables, arrays and complex data structures. The DIM and PARAM statements are almost identical, the difference being that DIM statements are used to declare storage used exclusively within the procedure, and the PARAM statement is used to declare variables received from another calling procedure.

When do you need to use the DIM statement? You do not need to for simple variables of type REAL, because this is the default format for undeclared variables. You also do not need to for 32-character STRING type variables (any name ending with a "$" is automatically assigned this type). Even though you do not have to declare variables in these two cases, you may want to anyway to improve your program's internal documentation. The things you **must** declare are:

1. Any simple variables of type BYTE, INTEGER or BOOLEAN.
2. Any simple STRING variables shorter or longer than 32 characters.
3. Arrays of any type.
4. Complex data structures of any type.

The TYPE statement does not really create variable storage. Its purpose is to describe a new data structure type that can be used in DIM or PARAM statements in addition to the five basic data types built-in to BASIC09. Therefore, the TYPE statement is only used in programs that utilize complex data structures.

DIM Statement

Syntax: DIM <var> [, <var>] : <type> [;<var> [, <var>] : <type>]

FUNCTION: The DIM statement is used to declare simple variables, arrays, or complex data structures of the five basic types or any user-defined type. During compilation, BASIC09 assigns storage required for all variables declared in DIM statements.

Declaring Simple Variables

Simple variables are declared by using the variable name in a DIM statement without a subscript. If variables are not explicitly declared, they are automatically assumed to be REAL or if the variable name ends with a "$" character, STRING[32] . Therefore, all simple variables of other types must be explicitly declared. For example:

```
DIM logical:BOOLEAN
```

Several variables can be declared in sequence by seperating each variable with a comma:

```
DIM a,b,c: STRING
```

In addition, several different types can be declared in a single DIM statement by using a ";" to separate different types:

```
DIM a,b,c:INTEGER; n,m:decimal; x,y,z:BOOLEAN
```

In this example a, b and c are type INTEGER, n and m are type "decimal" (a user-defined type), and x, y and z are type BOOLEAN.

String variables are declared the same way except that an optional maximum string length can be specified. If a length is not explicitly given, 32 characters are assumed:

```
DIM name:STRING[40]; address,city:STRING; zip:REAL
```

In this case "name" is a string variable of 40 characters maximum, "address" and "city" are string variables of 32 characters each, and "zip" is a real variable.

Array Declarations

Arrays can have one, two or three dimensions. The DIM statement format is the same as for simple variables except each name is followed by a subscript(s) to indicate its size. The maximum subscript size is 2,147,483,647, although memory may limit this. Simple variable and array declarations can be mixed in the same DIM statement:

```
DIM a(10),b(20,30),c:INTEGER; x(5,5,5):STRING[12]
```

In the example above, "a" is an array of 10 integers, "b" is a 20 by 30 table of integers, "c" is a simple integer variable, and "x" is a matrix of 12-character strings.

Arrays can be any basic or user-defined type. By declaring arrays of user-defined types, structures of arbitrary complexity and shape can be generated.

The following is an example declaration that generates a doubly-linked list of character strings. Each element of the array consists of the string containing the data and two integer "pointers".

```
TYPE link_pointers = fwd,back: INTEGER
TYPE element = info: STRING[64]; ptr: link_pointers
DIM list(100): element; index: INTEGER

(* make a circular list *)
BASE0
FOR index := 0 TO 99
  list(index).info := "secret message " + STR$(index)
  list(index).ptr.fwd := index+1
  list(index).ptr.back := index-1
NEXT index
(* fix the ends *)
list(0).ptr.back := 99
list(99).ptr.fwd := 0

(* Print the list *)
index=0
REPEAT
  PRINT list(index).info
  index := list(index).ptr.fwd
UNTIL index=0
END
```

PARAM Statement

SYNTAX: PARAM <var> [, <var>] : <type> [;<var> [, <var>] : <type>]

FUNCTION: PARAM is almost identical to the DIM statement, but it does not create variable storage. Instead, it describes what parameters the "called" procedure expects to receive from the "calling" procedure.

The programmer must insure that the total size of each parameter conforms to the amount of storage expected for each parameter in the called procedure as specified by the PARAM statement.

BASIC09 checks the size of each parameter, but it DOES NOT CHECK TYPE. The programmer should ensure that the parameters evaluated in the RUN statement and sent to the called procedure agree exactly with the PARAM statement specification with respect to: the number of parameters, their order, size, shape and type.

Because type-checking is not performed, if you understand how the system operates, you can make the parameter passing operation perform useful but normally illegal type conversions of identically-sized data structures. For example, passing a string of 80 characters to a procedure expecting a BYTE array having 80 elements will assign the numeric value of each character in the string to the corresponding element of the byte array.

TYPE Statement

SYNTAX: TYPE <typename> := <type decl>

FUNCTION: This statement is used to define new data types. New data types are defined as a "vector" (a one-dimensional array) of previously defined types. This structure differs from an array in that the various elements may be of different types, and the elements are accessed by field name instead of an array index. The syntax of the <type decl> conforms to the syntax of the DIM and PARAM statements. For example:

```
TYPE cust_recd := name,address(3):STRING; balance
```

This example creates a new data type called "cust_recd" which has three named fields: a field called "name" which is a string, a field called "address" which is a vector of three strings, and a field called "balance" which is a (default) REAL value.

The TYPE statement can include previously defined types so that very complex structures can be created such as lists, trees, etc. This statement does not create any variable storage itself; the storage is created when the newly defined type is used in a DIM statement.

The example shown below creates an array having 250 elements of type "cust_recd" that was defined above:

```
DIM customer_file(250):cust_recd
```

To access elements of the array in assignment statements, the field name is used as well as the index:

```
name$ = customer_file(35).name
customer_file(N+1).address(3) = "New York, NY"
customer_file(X).balance= 125.98
```

The complex structure allows creation of data types appropriate to the job at hand by providing more natural organization and association of data. Additionally, the position of the desired element is known and defined at compilation time and need not be calculated at run time, unlike arrays, and can therefore be accessed faster than arrays.

Type Structure Size

Occasionally it is necessary to know the exact size of a data structure. Each basic data type is stored in a certain format:

TYPE	MEMORY FORMAT
BYTE	One Byte
INTEGER	Four Bytes
REAL	Eight Bytes
STRING	One Byte / Character
BOOLEAN	One Byte

While it would seem easy to just add the components' sizes together to find the size of the entire structure, it is not that simple. BASIC09 will only start INTEGERS, REALS and complex data structures on an even word boundary. Additionally, complex data structures will be of even word length. Therefore BASIC09 will "pad" certain structures with an empty byte to accomplish this.

For example, the size of the following structure's individual components add up to 9 bytes:

```
TYPE junk = a : BYTE; b,c : INTEGER
```

The actual size of this structure is 10 bytes. BASIC09 has inserted a byte between "a" and "b".

The following examples show different data structures and their corresponding size:

STRUCTURE	SIZE
TYPE demo = a : BYTE; b : REAL; c : BYTE	12
TYPE junk = d(3) : STRING [3]; e : BOOLEAN	10
TYPE crap = f : BYTE; g : demo; h : BYTE; i : junk	26

If you are ever in doubt of the size of a data structure, you can use the function: SIZE (<name>). <name> is the name of the variable, array or structure. The size is returned as the number of bytes of the structure.

end of chapter 10

FILES AND UNIFIED INPUT/OUTPUT

A file is a logical construct for storing a sequence of data in memory. A file should always be named for convenience in use and storage. File data may be pure binary data, textual data (ASCII characters) or any other useful information.

Hardware input/output ("I/O") devices used by OS-9 also work like files. You can generally use any I/O facility regardless of whether you are working with disk files or I/O devices (such as printers). This single interface is standard for any device and simple communication facilities allow any device to be used with any other device. This concept is known as "unified I/O".

NOTE: Unified I/O can benefit routine programming. For example: file operations can be debugged by communicating with a terminal or printer instead of a storage device, and procedures which normally communicate with a terminal can be tested with data coming from and sent to a storage device.

BASIC09 normally works with two types of files: sequential files and random-access files.

A sequential file sends or receives textual data only in the order in which it is received. It is not generally possible to start over at the beginning of a sequential file once a number of bytes have been accessed. Many I/O devices such as printers are necessarily sequential.

A sequential file contains only valid ASCII characters; the READ and WRITE commands perform format conversion similar to that done automatically in INPUT and PRINT commands. A sequential file contains record-delimiter characters (carriage return) which separate the data created by different WRITE operations. Each WRITE command will send a complete sequential-file record, which is a variable number of characters terminated by a carriage return. Each READ will read all characters up to the next carriage return.

A random-access file sends and receives data in binary form (by the PUT or GET statements) exactly as it is internally represented in BASIC09. This minimizes both the time involved in converting the data to and from ASCII representation as well as reducing the file space required to store the data.

It is possible to PUT and GET individual bytes or a substructure of many bytes (in a complex structure). The GET structure statement merely recovers the number of bytes associated with that type of structure. It is possible to move to a particular byte in a random-access file by using **SEEK** to find it and using PUT or GET sequentially from that point.

In general, "SEEK #path,0" is equivalent to the REWIND which is used in some forms of BASIC. Since the random-access file contains no record-separators to indicate the size of particular elements of the file, the programmer should use the SIZE function to determine the size of a single element, then use SEEK to move to the desired element within the file.

A new file is created on a storage device by executing **CREATE**. Once a file exists, the **OPEN** command is used to notify the operating system to set up a channel to the desired device and return that path number to the BASIC09 program. This channel number is then used in file-access operations (e.g., READ, WRITE, GET, PUT, SEEK, etc.). When the programmer is finished with the file, it should be closed to assure that the file system has updated all data back onto magnetic media.

I/O PATHS

A "path" is a description of a "channel" through which data flows to or from a given program or device. When a path is created, OS-9 returns a unique number to identify the path in subsequent file operations. This "path number" is used by the I/O statements to specify the file to be used.

NOTE: In order for data to flow to or from a device, there must be an associated OS-9 device driver. For detailed information on device drivers, consult your "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL".

Three path numbers have special meanings because they are "standard I/O paths" representing BASIC09's interactive input/output. These are automatically "opened" for you and should not be closed except in very special circumstances. The standard I/O path numbers are:

0 Standard Input (Keyboard)
1 Standard Output (Terminal Display)
2 Standard Error/Status (Terminal Display)

The table below is a summary of the I/O statements within BASIC09 and their general usage. This reflects typical usage; most statements can be used with any I/O device or file. Sometimes certain statements are used in unusual ways by advanced programmers to achieve certain special effects.

Statement	Generally Used With	Data Format (File Type)
INPUT	Keyboard (interactive input)	Text (Sequential)
PRINT	Terminals, Printers	Text (Sequential)
OPEN	Disk Files and I/O Devices	Any
CREATE	Disk Files and I/O Devices	Any
CLOSE	Disk Files and I/O Devices	Any
DELETE	Disk Files	Any
SEEK	Disk Files	Binary (Random)
READ	Disk Files	Text (Sequential)
WRITE	Disk Files	Text (Sequential)
GET	Disk Files and I/O Devices	Binary (Random)
PUT	DISK Files and I/O Devices.	Binary (Random)

INPUT Statement

SYNTAX: INPUT [#<path num>,] ["<prompt>",] <input list>

FUNCTION: INPUT accepts input during the execution of a program. Input is normally read from the standard input device (terminal) unless an optional path number is given. When the INPUT statement is encountered, execution is suspended and a "?" prompt is displayed.

The INPUT statement is really both an input and output statement. If the optional prompt string is given, it is displayed instead of the normal "?" prompt. Therefore, if a path other than the default standard input path is used, the path should be open in UPDATE mode. This makes the INPUT statement dangerous if used on disk files. Unless you like prompts in your data, use READ.

The data entered is assigned in order to the variable names as they appear in the input list. The variables can be of any basic type, and the input data must be of the same or compatible type. The line is terminated by a carriage return. There must be at least as many input items given as variables in the input list. The length of the input line can not exceed 256 characters.

If any error occurs (type mismatch, insufficient amount of data, etc.), the entire input line must then be reentered. The following message is displayed, followed by a new prompt:

```
**INPUT ERROR - RETYPE**
```

The INPUT statement uses OS-9's line input function (READLN) which performs line editing such as backspace, delete, end-of-file, etc. To perform input WITHOUT editing, use the GET statement.

To test if data is available from the keyboard without "hanging" the program, use the "INKEY" function that returns the number of characters in the data buffer.

EXAMPLES:

```
INPUT number,name$,location

INPUT #14, "What is your selection", choice

INPUT "What's your name? ",name$;
```

To read a single character (without editing) from the terminal:

```
DIM char:STRING[1]
GET #0,char
```

PRINT Statement

SYNTAX: **PRINT** <output list>
PRINT #<path num>, <output list>
PRINT USING <str expr>, <output list>
PRINT #<path num>, **USING** <str expr>, <output list>

FUNCTION: PRINT outputs the values of the items given in the output list to the standard output device (path #1, the terminal) unless another path number is specified.

The output list consists of one or more items separated by a comma or semicolon delimiter. Each item can be a constant, variable or expression of any basic type. The PRINT statement evaluates each item and converts the result to corresponding ASCII characters which are then displayed.

If the separator character following the item is a semicolon, the next item will be displayed without any intermediate spacing. If a comma is used, spaces will be output so the next item starts at the next tab zone. The tab zones are 16 characters in length starting at the beginning of the line. If the line is terminated by a semicolon, the usual carriage return following the output line is inhibited.

"**TAB**(<expr>)" can be used as an item in the output list. This function causes the next item in the output list to start in the specified column (<expr>). If the output line is already past the desired tab position, the TAB is ignored.

A related function, "**POS**", can be used in a program to determine the output position at any given time. The output columns are numbered from one to a maximum of 255. The size of BASIC09's output buffer varies according to stack size at the moment.

The **PRINT USING** form of this statement is described at the end of this chapter.

EXAMPLES:

```
PRINT value,temp+(n/2.5),location$

PRINT #printer_path,"The result is "; n

PRINT "what is " + name$ + "'s age? ";

PRINT "index: ";i;TAB(25);"value:  ";value

(* print an 80-character line of all dashes *)
REPEAT
  PRINT "-";
UNTIL POS >= 80
PRINT
```

OPEN Statement

SYNTAX: OPEN #<path num>,"<pathlist>" {: <access mode>}

FUNCTION: OPEN issues a request to OS-9 to open an I/O path to an existing file or device.

The path number specified can be a variable. If so, it must be dimensioned as type INTEGER or BYTE and is used to "receive" the "path number" assigned to the path by OS-9. The path number is used to reference the specific file/device in subsequent input/output statements.

The pathlist must be within quotes. A string variable is allowed, and will be evaluated and passed to OS-9 as the descriptive pathlist.

The OPEN statement may also specify the path's desired "access mode" which can be READ, WRITE, UPDATE, EXEC or DIR. The access mode defines which direction I/O transfers will occur. If no access mode is specified, UPDATE is assumed and both reading and writing are permitted. The DIR mode allows OS-9 directory-type files to be accessed but should NOT be used in combination with WRITE or UPDATE modes. The EXEC mode causes the current execution directory to be used instead of the current data directory. For more information on files access modes, refer to the "OS-9/68000 OPERATING SYSTEM USER'S MANUAL".

EXAMPLES:

```
DIM printer_path:BYTE; name:STRING[24]
name="/p"
OPEN #printer_path,name:WRITE
PRINT #printer_path,"Mary had a little lamb"
CLOSE #printer_path


DIM inpath:INTEGER
dev$="/winchester/"
INPUT name$
OPEN #inpath,dev$+name$:READ

OPEN #path:userdir$:READ+DIR

OPEN #path,name$:WRITE+EXEC
```

CREATE Statement

SYNTAX: CREATE #<path num>,"<pathlist>" {: <access mode>}

FUNCTION: The CREATE statement is used to create a new file on a multifile mass storage device such as disk or tape. If the device is not of multifile type, this statement works like an "OPEN" statement.

The path number may be a variable. The variable name is used to "receive" the path number assigned by OS-9 and must be of BYTE or INTEGER type. The pathlist must be within quotes. A string variable is allowed, and will be evaluated and passed to OS-9 as the descriptive pathlist.

The "access mode" defines the direction of subsequent I/O transfers and should be either WRITE or UPDATE. "UPDATE" mode allows the file to be either read or written.

OS-9 has a single file type that can be accessed both sequentially OR at random. Files are byte-addressed, so no explicit "record" length need be given (see GET and PUT statements). When a new file is created, it has an initial length of zero. Files are expanded automatically by PRINT, WRITE or PUT statements that write beyond the current "end of file".

EXAMPLES:
```
CREATE #trans,"transactions":UPDATE

CREATE #spool,"/user4/report":WRITE

CREATE #outpath,name$:UPDATE+EXEC
```

CLOSE Statement

SYNTAX: CLOSE #<path num> [,#<path num>]

FUNCTION: The CLOSE statement notifies OS-9 that one or more I/O paths are no longer needed. The paths are specified by their number(s). If the closed path used a non-sharable device (such as a printer), the device is released and can be assigned to another user. The path must have been previously established by means of the OPEN or CREATE statements.

WARNING: Paths #0, #1 and #2 should never be closed unless the user immediately opens a new path to take over the Standard Path number.

EXAMPLES:

```
CLOSE #master,#trans,#new_master

CLOSE #5,#6,#9

CLOSE  #1           \(* closes standard output path *)
OPEN #path,"/T1"    \(* Permanently redirects Std Output *)

CLOSE #0             \(* closes standard input path *)
OPEN #path,"/TERM"  \(* Permanently redirects Std Input *)
```

DELETE Statement

SYNTAX: DELETE <pathlist>

This statement is used to delete a mass storage file. The file's name is removed from the directory and all its storage is deallocated, so any data in the file is permanently lost. The pathlist must be within quotes. A string variable is allowed, and will be evaluated and passed to OS-9 as the descriptive pathlist.

The user must have write permission for the file to be deleted. See the "OS-9/68000 Operating System User's Manual" for more information.

EXAMPLES:

```
DELETE "/D0/old_junk"

name$="file55"
DELETE name$
DELETE "/D2/"+name$      (deletes file named "/D2/file55")
```

SEEK Statement

SYNTAX: SEEK #<path num>,<real expr>

FUNCTION: SEEK changes the file pointer address of a mass storage file, which is the address of the next data byte(s) that are to be read or written. Therefore, this statement is essential for random access of data in files using the GET and PUT statements.

The #<path num> may be an expression that specifies the path number of the file and must evaluate to a byte value. The <real> expression specifies the desired file pointer address, and must evaluate to an INTEGER or REAL value in the range 0 <= result <= 2,147,483,647. Any fractional part of the result is truncated. Of course the actual maximum file size depends on the capacity of the device.

Although SEEK is normally used with random-access files, it can be used to "rewind" sequential files. For example:

```
SEEK #path,0
```

This is the same as a "rewind" or "restore" function. This is the only form of the SEEK statement that is generally useful for files accessed by READ and WRITE statements. These statements use variable-length records, so it is difficult to know the address of any particular record in the file.

EXAMPLES:

```
SEEK #fileone,filptr*2

SEEK #outfile,208894

SEEK #inventory,(part_num - 1) * SIZE(inv_rcd)
```

READ Statement

SYNTAX: READ #<path num>,<input list>

FUNCTION: The READ statement causes input data in ASCII character format to be read from a file or device.

The #<path num> may be an expression which specifies a path number. The path number must have been previously opened by an OPEN or CREATE statement in READ or UPDATE access mode (except the standard input path #0). Data is read starting at the path's current file pointer address which is updated as data is read.

READ calls OS-9 to read a variable length ASCII record. Individual data items within the record are converted to BASIC09's internal binary format. These results are assigned in order to the variables given in the input list. The input data must match the number and type of the variables in the input list.

The individual data items in the input record are separated by ASCII null characters. Numeric items can also be delimited by commas or space characters. The input record is terminated by a carriage return character.

EXAMPLES:

```
READ #inpath,name$,address$,city$,state$,zip

PRINT #1,"height,weight? "
READ #0,height,weight
```

NOTE: READ is also used to read lists of expressions in the program. See the DATA statement section for details.

WRITE Statement

SYNTAX: WRITE #<path num>,<output list>

FUNCTION: The WRITE statement writes data in ASCII character format on a file/device. The first expression specifies the number of a path that was previously opened by an OPEN or CREATE statement in WRITE or UPDATE mode.

The output list consists of one or more expressions separated by commas. Each expression can evaluate to any expression type. The result is then converted to an ASCII character string and written on the specified path beginning at the present file pointer which is updated as data is written.

If the output list has more than one item, ASCII null characters ($00) are written between each output string. The last item is followed by a carriage return character.

Note that this statement creates variable-length ASCII records.

EXAMPLES: WRITE #outpath,cat,dog,mouse

```
WRITE #xfile,LEFT$(A$,n);count/2
```

GET Statement / PUT Statement

SYNTAX: **GET** #<path num>,<struct name>
PUT #<path num>,<struct name>

FUNCTION: The GET and PUT statements read and write fixed-size binary data records to files or devices, respectively. These are the primary I/O statements used for random access input and output.

The path number may be an expression which is evaluated and used as the number of the I/O path which must have previously been opened by an OPEN or CREATE statement. Paths used by PUT statements must have been opened in WRITE or UPDATE access modes, and paths used by GET statements must be in READ or UPDATE mode.

The statement uses exactly one name which can be the name of a variable, array or complex data structure. Data is written from, or read into, the variable or structure named. The data is transferred in BASIC09's internal binary format without conversion which affords very high throughput compared to READ and WRITE statements. Data is transferred beginning at the current position of the path's file pointer (see SEEK statement) which is automatically updated.

OS-9's file system does not inherently impose record structures on random-access files. All files are considered to be continuous sequences of addressable binary bytes. A byte or group of bytes located anywhere in the file can be read or written in any order. Therefore the programmer is free to use the basic file access system to create any record structure desired.

Record I/O in BASIC09 is associated with data structures defined by DIM and TYPE statements. The GET and PUT statements write entire data structures or parts of data structures. A PUT statement, for example, can write a simple variable, an entire array, or a complex data structure in one operation.

To illustrate how this works, here is an example based on a simple inventory system that requires a random access file having 100 records. Each record must include the following information: the name of the item (a 25-byte character string), the item's list price and cost (both real numbers), and the quantity on hand (an integer).

First it is necessary to use the TYPE statement to define a new data type that describes such a record:

```
TYPE inv_item=name:STRING[25];list,cost:REAL;qty:INTEGER
```

This statement describes a new record type called "inv_item" but does not cause variable storage to be assigned for it. The next step is to create two data structures: an array of 100 "records" of type "inv_item" to be called "inv_array" and a single working record called "work_rec":

```
DIM inv_array(100):inv_item
DIM work_rec:inv_item
```

To calculate the total size of each record, you can manually count the number of bytes assigned for each type or use the built in SIZE function. Manually calculating record size can become complicated and error-prone (See the TYPE statement definition in the previous chapter). Also, any change in a TYPE definition could require recalculation. Fortunately, BASIC09 has a built-in function:

```
SIZE(<name>)
```

This returns the number of bytes assigned to any variable, array, or complex data structure. In our example, SIZE(work_rec) will return the number 46, and SIZE(inv_array) will return 4600. The size function is often used in conjunction with the SEEK statement to position a file pointer to a specific record's address.

The procedure below creates a file called "inventory" and initializes it with zeroes and nulls:

```
PROCEDURE makefile
TYPE inv_item = name:STRING[25];list,cost:REAL;qty:INTEGER
DIM inv_array(100):inv_item
DIM work_rec:inv_item
DIM path:byte
CREATE #path,"inventory"
work_rec.name = ""
work_rec.list := 0.
work_rec.cost := 0.
work_rec.qty := 0
FOR n = 1 TO 100
  PUT #path,work_rec
NEXT n
END
```

Notice that the assignment statements reference each named "field" of work_rec by name, but the PUT statement references the record as a whole.

The subroutine below asks for a record number, then asks for data and writes it on the file at the specified record:

```
INPUT "Record number ?",recnum
INPUT "Item name? ",work_rec.name
INPUT "List price? ",work_rec.list
INPUT "Cost price? ",work_rec.cost
INPUT "Quantity? ",work_rec.qty
SEEK #path, (recnum - 1) * SIZE(work_rec)
PUT #path,work_rec
```

The routine below uses a loop to read the entire file into the array "inv_array":

```
SEEK #path,0 \ (* "rewind" the file *)
FOR K = 1 TO 100
  GET #path,inv_array(k)
NEXT k
```

Because ENTIRE STRUCTURES can be read, we can eliminate the FOR/NEXT loop and do exactly the same thing by:

```
SEEK #path,0
GET #path,inv_array
```

The above example is a very simple case, but it illustrates the combined power of BASIC09 complex data structures and the random access I/O statements. When fully exploited, this system has the following important characteristics:

1. It is self-documenting. You can clearly see what a program does, because structures have descriptive named sub-structures.

2. It is extremely fast.

3. Programs are simplified and require fewer statements to perform I/O functions than in other BASIC languages.

4. It is versatile. You can read or write almost any kind of data on any file, including files created by other programs or languages by creating appropriate data structures.

These advantages are possible because a single GET or PUT statement can move any amount of data, organized any way you want.

INTERNAL DATA STATEMENTS

DATA Statement / READ Data Statement / RESTORE Statement

```
SYNTAX:  READ <input list>
         DATA <expr> , { <expr> }
         RESTORE [ <line number> ]
```

FUNCTION: These statements provide an efficient way to build constant tables within a program. DATA statements provide values. READ statements assign the values to variables. RESTORE statements can be used to set which data statement is to be read next.

The DATA statements have one or more expressions separated by commas. They can be located anywhere in a program. The expressions are evaluated each time the data statements are read and can evaluate to any type.

The following examples demonstrate the DATA statement:

```
DATA 1.1,1.5,9999,"CAT","DOG"
DATA SIN(temp/25), COS(temp*PI)
DATA TRUE,FALSE,TRUE,TRUE,FALSE
```

The READ statement has a list of one or more variable names. When executed, it reads "input" by evaluating the current expression in the current data statement. The result must match the type of the variable. When all the expressions in a DATA statement have been evaluated, the next DATA statement (in sequential order) is used. If there are no more DATA statements following, processing "wraps around" to the first data statement in the program.

The RESTORE statement used without a line number causes the first DATA statement in the program to be used next. If it is used with a line number, the data statement having that line number is used next.

EXAMPLE:

```
        DATA 1,2,3,4
        DATA 5,6,7,8
100     DATA 9,10,11,12
        FOR N := 1 TO X
          READ ARRAY(N)
        NEXT N
        RESTORE 100
        READ A,B,C,D
```

FORMATTED OUTPUT: THE PRINT USING STATEMENT

BASIC09 has a powerful output editing capability useful for report generation and other applications where formatted output is required. The output editing uses the **PRINT USING** statement:

SYNTAX: **PRINT** [<path#>] **USING** <str expr> , <output list>

The optional path number expression can be used to specify the path number of any output file or device. If it is omitted, the output is written to the standard output path (usually the terminal).

The string expression is evaluated and used as a "format specification". This contains specific formatting directives for each item in the "output list". **BLANKS ARE NOT ALLOWED IN FORMAT STRINGS!**

The items in the output list can be constants, variables, or expressions of any atomic type. As each output item is processed, it is matched up with a specification in the format list. The type of each expression result must be compatible with the corresponding format specification. If there are fewer format specifications than items in the output list, the format specification list is repeated again from its beginning as many times as necessary.

A format string has one or more format specifications which are separated by commas. There are two kinds of specifications: ones that control output editing of an item from the output list and ones that cause an output function by themselves (such as tabbing and spacing). There are six basic output editing directives. Each has a corresponding one-letter identifier:

R	real format
E	exponential format
I	integer format
H	hexadecimal format
S	string format
B	boolean format

The identifier letter is followed by a constant number called the "field width". This number indicates the exact number of print columns the output is to occupy and must allow for the data **AND** "overhead" character positions such as sign characters, decimal points, exponents, etc.

Some formats have additional mandatory or optional parameters that control subfields or select editing options. One of these options is "justification" which specifies whether the output is to center or "line up" the output field on the left or right side. Fields are commonly right-justified in reports because it arranges them into columns with decimal points aligned in the same position.

The abbreviations and symbols used in the syntax specifications are:

w	Total field width:	1 <= w <= 255
f	fraction field:	1 <= w <= 9
j	OPTIONAL justification:	< (left) > (right) (center)

REAL FORMAT

SYNTAX: Rw.fj

REAL format can be used for numbers of types REAL, INTEGER or BYTE. The total field width specification must include two overhead positions for the sign and decimal point. The "f" specifies how many fractional digits to the right of the decimal point are to be displayed. If the number has more significant digits than the field allows for, the undisplayed places are used to round the displayed digits. For example:

```
PRINT USING "R8.2", 12.349      Output:  12.35
```

The justification modes are:

`<` left justify with leading sign and trailing spaces. (default if justification mode omitted)

`>` right justify with leading spaces and sign.

right justify with leading spaces and trailing sign (financial format)

EXAMPLE:

```
PROCEDURE printdemo
PRINT USING "R8.2<",5678.123
PRINT USING "R8.2>",12.3
PRINT USING "R8.2<",-555.9
PRINT USING "R10.2 ",-6722.4599
PRINT USING "R5.1",9999999
END

RUN printdemo
 5678.12
   12.30
-555.90
  6722.46-
*****
```

EXPONENTIAL FORMAT

SYNTAX: Ew.fj

Exponential format prints numbers of types REAL, INTEGER, or BYTE in the scientific notation format using a mantissa and decimal exponent. The syntax and behavior of this format is similar to the REAL format except the "w" (field width) must allow for six overhead positions for the mantissa sign, decimal point, and exponent characters.

The justification modes are:

- < left justify with leading sign and trailing spaces. (default if justification mode omitted)
- > right justify with leading spaces and sign.

EXAMPLE:

```
PROCEDURE expodemo
PRINT USING "E12.3",1234.567;
PRINT USING "E13.6>",-0.001234;
PRINT USING "E12.2",1234567
END

RUN expodemo
 1.235E+03  -1.234000E-03 1.23E+06
```

INTEGER FORMAT

SYNTAX: Iwj

FUNCTION: This format is used to display numbers of types INTEGER or BYTE, and REAL numbers that are within range for automatic type conversion. The "w" (field width) must allow for one position overhead for the sign.

The justification modes are:

< left justify with leading sign and trailing spaces. (default if justification mode omitted)

> right justify with leading spaces and sign.

right justify with leading spaces, sign and zeroes.

EXAMPLE:

```
PROCEDURE intdemo
PRINT USING "I4<",10
PRINT USING "I4>",10
PRINT USING "I4 ",10
END

RUN intdemo
 10
  10
 010
```

HEXADECIMAL FORMAT

SYNTAX: Hwj

FUNCTION: This format can be used to display the internal binary representation of ANY data type, using hexadecimal characters. The "w" (field width) specifies the total field size. If the hexidecimal representation to be displayed is shorter than the field size, it is padded with spaces according to the justification mode. If a representation of a numeric is too long, it will be truncated on the left side (see the second line in the procedure and its matching output below). Representations of STRINGS that are not allowed a large enough field are truncated from the right.

The number of bytes of memory used to represent data varies according to type. The following specifications are the minimum required field widths to display the entire hexidecimal representation for each data type:

H2	boolean, byte (one byte)
H8	integer (four bytes)
H16	real (eight bytes)
Hn*2	string of length n

The justification modes are:

< left justify with trailing spaces (default)

> right justify, leading spaces

  center justify

EXAMPLES:

```
PROCEDURE hexdemo
PRINT USING "H8",100
PRINT USING "H4",100
PRINT USING "H16",1.5
PRINT USING "H8,H10,H20>","ABC",1,1.5
END

RUN hexdemo
00000064
0064
01C0000000000000
414243  00000001       01C0000000000000
```

STRING FORMAT

SYNTAX: Sw*j*

FUNCTION: This format is used to display string data of any length. The "w" (field width) specifies the total field size. If the string to be displayed is shorter than the field size, it is padded with spaces according to the justification mode. If it is too long, it will be truncated on the right side.

The justification modes are:

<	Left justify (default if mode omitted)
>	right justify
	Center justify

EXAMPLES:

```
PROCEDURE stringdemo
PRINT USING "S9<","HELLO"
PRINT USING "S9>","HELLO"
PRINT USING "S9 ","HELLO"
END

RUN stringdemo
HELLO
    HELLO
  HELLO
```

BOOLEAN FORMAT

SYNTAX: Bwj

FUNCTION: This format is used to display boolean data. The result of the boolean expression is converted to the strings "TRUE" and "FALSE". The "w" (field width) specifies the total field size. If the string to be displayed is shorter than the field size, it is padded with spaces according to the justification mode. If it is too long, it will be truncated on the right side.

The justification modes are:

<	Left justify (default if mode omitted)
>	right justify
	Center justify

EXAMPLE:

```
PROCEDURE booldemo
PRINT "IS 10<4 ?"
PRINT USING "B9>", 10<4
PRINT "IS 4<10 ?"
PRINT USING "B9", 4<10
END

RUN booldemo
IS 10<4 ?
    FALSE
IS 4<10 ?
TRUE
```

CONTROL SPECIFICATIONS

Control specifications are useful for horizontal formatting of the output line. They are not matched with items in the output list and can be used freely. The control formats are:

Tn	Tab to column n
Xn	Space n columns
'str'	Print constant string. The string must not include single or double quotes, backslash, or carriage return characters.

WARNING: Control specifications at the end of the format specification list will **NOT** be processed if all output items have been exhausted.

EXAMPLE:
```
PROCEDURE demo
PRINT USING "'addr',X2,H4,X2,'data',X2,H2",1000,100
END

RUN demo
addr  03E8  data  64
```

REPEAT GROUPS

Many times, identical sequences of specifications are repeated in format specification lists. The repeated groups can be enclosed in parentheses and preceded by a repeat count. These repeat groups can be nested.

EXAMPLES:

"2(X2,R10.5)" is the same as "X2,R10.5,X2,R10.5"

"2(I2,2(X1,S4))" is the same as "I2,X1,S4,X1,S4,I2,X1,S4,X1,S4"

end of chapter 11

USER NOTES

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL
APPENDIX A
SAMPLE PROGRAMS

Appendix A contains 24 example procedures that can be used immediately or studied and adapted for use within user procedures. The procedures are:

```
PROCEDURE fibonacci
  REM computes the first eleven Fibonacci numbers
  DIM x,y,i,temp:INTEGER

  x:=0 \y:=0
  FOR i=0 TO 10
    temp:=y

    IF i<>0 THEN
      y:=y+x
    ELSE y:=1
    ENDIF

    x:=temp
    PRINT i,y
  NEXT i


PROCEDURE fractions
  REM by T.F. Ritter
  REM finds increasingly-close rational approximations
  REM to the desired real value
  DIM m:INTEGER

  desired:=PI
  last:=0

  FOR m=1 TO 30000
    n:=INT(.5+m*desired)
    trial:=n/m
    IF ABS(trial-desired)<ABS(last-desired) THEN
      PRINT n; "/"; m; " = "; trial,
      PRINT "difference = "; trial-desired;
      PRINT
      last:=trial
    ENDIF
  NEXT m
```

```
PROCEDURE prinbi
  REM by T.F. Ritter
  REM prints the integer parameter value in binary
  PARAM n:INTEGER
  DIM i:INTEGER

  FOR i=31 TO 0 STEP -1
    IF n<0 THEN
      PRINT "1";
    ELSE PRINT "0";
    ENDIF
    n:=n+n
  NEXT i
  PRINT

  END


PROCEDURE hanoi
  REM by T.F. Ritter
  REM move n discs in Tower of Hanoi game
  REM See BYTE Magazine, Oct 1980, pg. 279

  PARAM n:INTEGER; from,to_,other:STRING[8]

  IF n=1 THEN
    PRINT "move #"; n; " from "; from; " to "; to_
  ELSE
    RUN hanoi(n-1,from,other,to_)
    PRINT "move #"; n; " from "; from; " to "; to_
    RUN hanoi(n-1,other,to_,from)
  ENDIF

  END
```

```
PROCEDURE roman
  REM prints integer parameter as Roman Numeral
  PARAM x:INTEGER
  DIM value,svalu,i:INTEGER
  DIM char,subs:STRING

  char:="MDCLXVI"
  subs:="CCXIII "
  DATA 1000,100,500,100,100,10,50,10,10,1,5,1,1,0

  FOR i=1 TO 7
    READ value
    READ svalu

    WHILE x>=value DO
      PRINT MID$(char,i,1);
      x:=x-value
    ENDWHILE

    IF x>=value-svalu THEN
      PRINT MID$(subs,i,1); MID$(char,i,1);
      x:=x-value+svalu
    ENDIF

  NEXT i
  END
```

```
PROCEDURE eightqueens
  REM originally by N. Wirth; here re-coded from Pascal
  REM finds the arrangements by which eight queens
  REM can be placed on a chess board without conflict
  DIM n,k,x(8):INTEGER
  DIM col(8),up(15),down(15):BOOLEAN
  BASE 0

  (* initialize empty board *)
  n:=0
  FOR k:=0 TO 7 \col(k):=TRUE \NEXT k
  FOR k:=0 TO 14 \up(k):=TRUE \down(k):=TRUE \NEXT k
  RUN generate(n,x,col,up,down)
  END

PROCEDURE generate
  PARAM n,x(8):INTEGER
  PARAM col(8),up(15),down(15):BOOLEAN
  DIM h,k:INTEGER \h:=0
  BASE 0

  REPEAT
    IF col(h) AND up(n-h+7) AND down(n+h) THEN
      (* set queen on square [n,h] *)
      x(n):=h
      col(h):=FALSE \up(n-h+7):=FALSE \down(n+h) := FALSE
      n:=n+1
      IF n=8 THEN
        (* board full; print configuration *)
        FOR k=0 TO 7
          PRINT x(k); "   ";
        NEXT k
        PRINT
      ELSE RUN generate(n,x,col,up,down)
      ENDIF

      (* remove queen from square [n,h] *)
      n:=n-1
      col(h):=TRUE \up(n-h+7):=TRUE \down(n+h):=TRUE
    ENDIF
    h:=h+1
  UNTIL h=8
  END
```

```
PROCEDURE electric
  REM re-programmed from "ELECTRIC"
  REM by Dwyer and Critchfield
  REM Basic and the Personal Computer (Addison-Wesley, 1978)
  REM provides a pictorial representation of the
  REM resultant electrical field around charged points
  DIM a(10),b(10),c(10)
  DIM x,y,i,j:INTEGER
  xscale:=50./78.
  yscale:=50./32.

  INPUT "How many charges do you have? ",n
  PRINT "The field of view is 0-50,0-50 (x,y)"
  FOR i=1 TO n
    PRINT "type in the x and y positions of charge ";
    PRINT i;
    INPUT a(i),b(i)
  NEXT i

  PRINT "type in the size of each charge:"
  FOR i=1 TO n
    PRINT "charge "; i;
    INPUT c(i)
  NEXT i

  REM visit each screen position
  FOR y=32 TO 0 STEP -1
    FOR x=0 TO 78
      REM compute field strength into v
      GOSUB 10
      z:=v*50.
      REM map z to valid ASCII in b$
      GOSUB 20
      REM print char (proportional to field)
      PRINT b$;
    NEXT x
    PRINT
  NEXT y
  END

10  v=1.
  FOR i=1 TO n
    r:=SQRT(SQ(xscale*x-a(i))+SQ(yscale*y-b(i)))
    EXITIF r=.0 THEN
      v:=99999.
    ENDEXIT
    v:=v+c(i)/r
  NEXT i
  RETURN
```

(continued on next page)

PROCEDURE ELECTRIC - CONTINUED

```
20   IF z<32 THEN b$:=" "
     ELSE
       IF z>57 THEN z:=z+8
       ENDIF
       IF z>90 THEN b$:="*"
       ELSE
         IF z>INT(z)+.5 THEN b$:=" "
         ELSE b$:=CHR$(z)
         ENDIF
       ENDIF
     ENDIF
     RETURN
```

```
PROCEDURE structst

  REM example of intermixed array and record structures
  REM note tnat structure d contains 200 real elements

  TYPE a=one(2):REAL
  TYPE b=two(10):a
  TYPE c=three(10):b
  DIM d,e:c

  FOR i=1 TO 10
    FOR j=1 TO 10
      FOR k=1 TO 2
        PRINT d.three(i).two(j).one(k)
        d.three(i).two(j).one(k):=0.
        PRINT e.three(i).two(j).one(k)
        PRINT
      NEXT k
    NEXT j
  NEXT i

  REM this is a complete structure assignment
  e:=d

  FOR i=1 TO 10
    FOR j=1 TO 10
      FOR k=1 TO 2
        PRINT e.three(i).two(j).one(k);
      NEXT k
    PRINT
    NEXT j
  NEXT i

  END
```

```
PROCEDURE wordlook
  REM Display a word at a user specifiede address
  REM By Andy Nicholson

  DIM address, datum: INTEGER
  INPUT "Enter word address:  ";address

  datum := PEEK(address)
  RUN prinbyte(datum)
  datum := PEEK(address + 1)
  RUN prinbyte(datum)
  datum := PEEK(address + 2)
  RUN prinbyte(datum)
  datum := PEEK(address + 3)
  RUN prinbyte(datum)
  END

PROCEDURE prinbyte
  REM print a byte as binary
  PARAM n: INTEGER
  DIM i: INTEGER

  n:= n*16777216
  FOR i = 7 TO 0 STEP -1
    IF n < 0 THEN
      PRINT "1"
    ELSE
      PRINT "0"
    ENDIF
    n:= n + 1
  NEXT i

  PRINT
  END
```

```
PROCEDURE qsort1
  REM quicksort, by T.F. Ritter
  PARAM bot,top,d(1000):INTEGER
  DIM n,m:INTEGER; btemp:BOOLEAN

  n:=bot
  m:=top


  LOOP  \REM each element gets the once over

    REPEAT  \REM this is a post-inc instruction
      btemp:=d(n)<d(top)
      n:=n+1
    UNTIL NOT (btemp)
    n:=n-1 \REM point at the tested element
  EXITIF n=m THEN
  ENDEXIT

    REPEAT  \REM this is a post-dec instruction
      m:=m-1
    UNTIL d(m)<=d(top) OR m=n
  EXITIF n=m THEN
  ENDEXIT

    RUN exchange(d(m),d(n))
    n:=n+1 \REM prepare for post-inc
  EXITIF n=m THEN
  ENDEXIT

  ENDLOOP


  IF n<>top THEN
    IF d(n)<>d(top) THEN
      RUN exchange(d(n),d(top))
    ENDIF
  ENDIF

  IF bot<n-1 THEN
    RUN qsort1(bot,n-1,d)
  ENDIF
  IF n+1<top THEN
    RUN qsort1(n+1,top,d)
  ENDIF

  END
```

(continued on next page)

```
PROCEDURE exchange
  PARAM a,b:INTEGER
  DIM temp:INTEGER

  temp:=a
  a:=b
  b:=temp

  END


PROCEDURE prin
  PARAM n,m,d(1000):INTEGER
  DIM i:INTEGER

  FOR i=n TO m
    PRINT d(i);
  NEXT i
  PRINT

  END


PROCEDURE sortest
REM This procedure is used to test Quicksort
REM It fills the array "d" with randomly generated
REM numbers and sorts them.
  DIM i,d(1000):INTEGER

  FOR i=1 TO 1000
    d(i):=INT(RND(100))
  NEXT i

  RUN prin(1,1000,d)

  RUN qsort1(1,1000,d)

  RUN prin(1,1000,d)

  END
```

The following procedures demonstrate multiple-precision arithmetic, in this case using five integers to represent a 40 decimal digit number, with eight fractional places.

```
PROCEDURE upadd
  REM a+b=>c:five_integer_number (T.F. Ritter)
  PARAM a(5),b(5),c(5):INTEGER
  DIM i,carry:INTEGER

  carry:=0
  FOR i=5 TO 1 STEP -1
    c(i):=a(i)+b(i)+carry
    IF c(i)>=100000000 THEN
      c(i):=c(i)-100000000
      carry:=1
    ELSE carry:=0
    ENDIF
  NEXT i

PROCEDURE upsub
  PARAM a(5),b(5),c(5):INTEGER
  DIM i,borrow:INTEGER

  borrow:=0
  FOR i=5 TO 1 STEP -1
  c(i):=a(i)-b(i)-borrow
  IF c(i)<0 THEN
  c(i):=c(i)+100000000
  borrow:=1
  ELSE borrow:=0
  ENDIF
  NEXT i

PROCEDURE uprint
  PARAM a(5):INTEGER
  DIM i:INTEGER; s:STRING

  FOR i=1 TO 5
  IF i=5 THEN PRINT ".";
  ENDIF
  s:=STR$(a(i))
  PRINT RIGHT$("00000000"+s,8);
  NEXT i
```

```
PROCEDURE upinput
  PARAM a(5):INTEGER
  DIM n,i:INTEGER
  DIM b$:STRING[64]

  INPUT "input ultra-precision number: ",b$
  b$:=TRIM$(b$)
  n:=SUBSTR(".",b$)

  IF n<>0 THEN
  a(5):=VAL(MID$(b$+"00000000",n+1,8))
  b$:=LEFT$(b$,n-1)
  ELSE a(5):=0
  ENDIF

  b$:="00000000000000000000000000000000"+b$
  n:=1+LEN(b$)
  FOR i=4 TO 1 STEP -1
  n:=n-8
  a(i):=VAL(MID$(b$,n,8))
  NEXT i

PROCEDURE uptoreal
  PARAM a(5):INTEGER; b:REAL
  DIM i:INTEGER

  b:=a(1)
  FOR i=2 TO 4
  b:=b*100000000
  b:=b+a(i)
  NEXT i
  b:=b+a(5)*.00000001

PROCEDURE ultra
  DIM one(5),two(5),out(5):INTEGER; r:REAL

  RUN upinput(one)
  RUN upinput(two)
  PRINT "add"
  RUN upadd(one,two,out)
  RUN uprint(out)
  PRINT
  RUN uptoreal(out,r)
  PRINT r
  PRINT "sub"
  RUN upsub(one,two,out)
  RUN uprint(out)
  PRINT
  RUN uptoreal(out,r)
  PRINT r
```

```
PROCEDURE Patch
(* Program to examine and patch any byte of a disk file *)
(* Written by L. Crane *)
DIM buffer(256):BYTE
DIM path,offset,modloc:INTEGER; loc:REAL
DIM rewrite:STRING
INPUT "pathlist? ",rewrite
OPEN #path,rewrite:UPDATE
LOOP
  INPUT "sector number? ",rewrite
EXITIF rewrite="" THEN ENDEXIT
  loc=VAL(rewrite)*256
  SEEK #path,loc
  GET #path,buffer
  RUN DumpBuffer(loc,buffer)
  LOOP
    INPUT "change (sector offset)? ",rewrite
  EXITIF rewrite="" THEN
    RUN DumpBuffer(loc,buffer)
  ENDEXIT
  EXITIF rewrite="S" OR rewrite="s" THEN ENDEXIT
    offset=VAL(rewrite)+1
    LOOP
    EXITIF offset>256 THEN ENDEXIT
      modloc=loc+offset-1
      PRINT USING "h4,' - ',h2",modloc,buffer(offset);
      INPUT ":",rewrite
    EXITIF rewrite="" THEN ENDEXIT
      IF rewrite<>" " THEN
        buffer(offset)=VAL(rewrite)
      ENDIF
      offset=offset+1
    ENDLOOP
  ENDLOOP
  INPUT "rewrite sector? ",rewrite
  IF LEFT$(rewrite,1)="Y" OR LEFT$(rewrite,1)="y" THEN
    SEEK #path,loc
    PUT #path,buffer
  ENDIF
ENDLOOP
CLOSE #path
BYE
```

(Continued on next page *)

PATCH - CONTINUED

```
PROCEDURE DumpBuffer
(* Called by PATCH *)
TYPE buffer=char(4):INTEGER
PARAM loc:REAL; line(16):buffer
DIM i,j:INTEGER
WHILE loc>65535. DO
  loc=loc-65536.
ENDWHILE
FOR j=1 TO 16
  PRINT USING "h4",FIX(INT(loc))+(j-1)*16;
  PRINT ":";
  FOR i=1 TO 4
    PRINT USING "X1,H8",line(j).char(i);
  NEXT i
  RUN printascii(line(j))
  PRINT
NEXT j


PROCEDURE PrintASCII
TYPE buffer=char(16):BYTE
PARAM line:buffer
DIM ascii:STRING; nextchar:BYTE; i:INTEGER
ascii=""
FOR i=1 TO 16
  nextchar=line.char(i)
  IF nextchar>127 THEN
    nextchar=nextchar-128
  ENDIF
  IF nextchar<32 OR nextchar>125 THEN
    ascii=ascii+" "
  ELSE
    ascii=ascii+CHR$(nextchar)
  ENDIF
NEXT i
PRINT "  "; ascii;
```

```
PROCEDURE MakeProc
  (* Generates an OS-9 command file to apply a command *)
  (* Such as copy, del, etc., to all files in a directory *)
  (* or directory system.  Author: L. Crane *)

  DIM DirPath,ProcPath,i,j,k:INTEGER
  DIM CopyAll,CopyFile:BOOLEAN
  DIM ProcName,FileName,ReInput,ReOutput,response:STRING
  DIM SrcDir,DestDir,DirLine:STRING[80]
  DIM Function,Options:STRING[60]
  DIM ProcLine:STRING[160]

  ProcName="CopyDir"
  Function="Copy"
  Options="-b=32k"

  REPEAT
    PRINT "Proc name ("; ProcName; ")";
    INPUT response

    IF response<>"" THEN
      ProcName=TRIM$(response)
    ENDIF

    ON ERROR GOTO 100
    SHELL "del "+ProcName
100 ON ERROR
    INPUT "Source Directory? ",SrcDir
    SrcDir=TRIM$(SrcDir)
    ON ERROR GOTO 200
    SHELL "del procmaker...dir"
200 ON ERROR
    SHELL "dir -u "+SrcDir+" >procmaker...dir"
    OPEN #DirPath,"procmaker...dir":READ
    CREATE #ProcPath,ProcName:WRITE
    PRINT "Function ("; Function; ")";
    INPUT response

    IF response<>"" THEN
      Function=TRIM$(response)
    ENDIF

    INPUT "Redirect Input? ",response

    IF response="y" OR response="Y" THEN
      ReInput="<" \ ELSE  \ReInput=""
    ENDIF

    INPUT "Redirect Output? ",response
```

MAKEPROC - CONTINUED

```
IF response="y" OR response="Y" THEN
  ReOutput=">" \ ELSE  \ReOutput=""
ENDIF

PRINT "Options ("; Options; ")";
INPUT response

IF response<>"" THEN
  Options=TRIM$(response)
ENDIF

INPUT "Destination Directory? ",DestDir
DestDir=TRIM$(DestDir)
WRITE #ProcPath,"t"
WRITE #ProcPath,"TMode .1 -pause"
READ #DirPath,DirLine
INPUT "Use all files? ",response
CopyAll=response="y" OR response="Y"

WHILE NOT(EOF(#DirPath)) DO
  READ #DirPath,DirLine
  i=LEN(TRIM$(DirLine))

  IF i>0 THEN
    j=1

    REPEAT
      k=j

      WHILE j<=i AND MID$(DirLine,j,1)<>" " DO
        j=j+1
      ENDWHILE

      FileName=MID$(DirLine,k,j-k)

      IF NOT(CopyAll) THEN
        PRINT "Use "; FileName;
        INPUT response
        CopyFile=response="y" OR response="Y"
      ENDIF

      IF CopyAll OR CopyFile THEN
        ProcLine=Function+" "+ReInput+SrcDir+"/"+FileName

        IF DestDir<>"" THEN
          ProcLine=ProcLine+" "+ReOutput+DestDir
          +"/"+FileName
        ENDIF
```

MAKEPROC - CONTINUED

```
        ProcLine=ProcLine+" "+Options
        WRITE #ProcPath,ProcLine
      ENDIF

      WHILE j<i AND MID$(DirLine,j,1)=" " DO
        j=j+1
      ENDWHILE

    UNTIL j>=i
  ENDIF
ENDWHILE

WRITE #ProcPath,"TMode .1 pause"
WRITE #ProcPath,"Dir e "+SrcDir

IF DestDir<>"" THEN
  WRITE #ProcPath,"Dir e "+DestDir
ENDIF

CLOSE #DirPath
CLOSE #ProcPath
SHELL "del procmaker...dir"
PRINT
INPUT "Another ? ",response
UNTIL response<>"N" AND response<>"n"

IF response<>"B" AND response<>"b" THEN
  BYE
ENDIF
```

```
*|-------------------------------------------------------------|
*|    SysCall  -  a subroutine for Basic09/68000
*|
*|    Called by:  RUN SysCall(Code,Regs)
*|
*|     Code  -  An integer contained the OS9 function code
*|     Regs  -  Register pack, must be at least 48 bytes for
*|                registers D0 thru A4
*|
          use       <oskdefs.d>

*|    Definition of Parameters on Stack
          org       0
Return    do.l      1               Return Address
Length1   do.l      1               Length of first parameter
Param2    do.l      1               Address of second parameter
Length2   do.l      1               Length of second parameter

          psect     SysCall,(Sbrtn<<8)!Objct,(ReEnt<<8)!1,0,0,SysCall

SysCall   cmpi.l    #2,d0           check parameter count
          bne.s     ParamErr        branch if error
          cmpi.l    #4,Length1(a7) is first parameter integer?
          bne.s     ParamErr        branch if not
          cmpi.l    #52,Length2(a7) 48 bytes of registers?
          blo.s     ParamErr        branch if not
*|    Now put the model on the stack
          move.w    #Modllen/2,d2   number of words for dbra
          lea       Model+Modllen(pc),a0 get address of model
          bra.s     SysC02          branch into loop
SysC01    move.w    -(a0),-(a7)     move a word
SysC02    dbra      d2,SysC01       continue if not done
          move.l    d1,a0           point to function code
          move.w    2(a0),2(a7)     set function code
*|    Get the registers
          movea.l   Param2+Modllen(a7),a5 get address of parameter
          movem.l   (a5)+,d0-d7/a0-a4 get register
          jsr       (a7)            call function
          movem.l   d0-d7/a0-a4,-(a5)
          lea.l     Modllen(a7),a7 clear stack
          rts

ParamErr  move.w    #E$Param,d1     get error code
          ori       #Carry,ccr      set carry
          rts

Model     os9       F$Fork          model system call to put on stack
          rts
Modllen   equ       *-Model
          ends
```

USER NOTES

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL
APPENDIX B
QUICK REFERENCE

SYSTEM MODE COMMANDS

$	DIGITS	EDIT	LOAD	RENAME
BYE	DIR	KILL	MEM	RUN
CHD	E	LIST	PACK	SAVE
CHX				

EDIT MODE COMMANDS

+	<cr>	c*	l*	r*
+*	<line #>	d	q	s
-	<space>	d*	r	s*
-*	c	l		

DEBUG MODE COMMANDS

$	DEG	LIST	RAD	TROFF
BREAK	DIR	PRINT	STATE	TRON
CONT	LET	Q	STEP	

BASIC09 RESERVED WORDS

ABS	DIR	INT	PEEK	SQR
ACS	DO	INTEGER	PI	SQRT
ADDR	ELSE	KILL	POKE	STEP
AND	END	LAND	POS	STOP
ASC	ENDEXIT	LEFT$	PRINT	STR$
ASN	ENDIF	LEN	PROCEDURE	STRING
ATN	ENDLOOP	LET	PUT	SUBSTR
BASE	ENDWHILE	LNOT	RAD	TAB
BOOLEAN	EOF	LOG	READ	TAN
BYE	ERR	LOG10	REAL	THEN
BYTE	ERROR	LOOP	REM	TO
CHAIN	EXEC	LOR	REPEAT	TRIM$
CHD	EXITIF	LXOR	RESTORE	TROFF
CHR$	EXP	MID$	RETURN	TRON
CHX	FALSE	MOD	RIGHT$	TRUE
CLOSE	FIX	NEXT	RND	TYPE
COS	FLOAT	NOT	RUN	UNTIL
CREATE	FOR	ON	SEEK	UPDATE
DATA	GET	OPEN	SGN	USING
DATE$	GOSUB	OR	SHELL	VAL
DEG	GOTO	PARAM	SIN	WHILE
DELETE	IF	PAUSE	SIZE	WRITE
DIM	INPUT		SQ	XOR
DIGITS				

BASIC09/68000 PROGRAMMING LANGUAGE USER'S MANUAL
APPENDIX B
QUICK REFERENCE

BASIC09 STATEMENTS

BASE 0	ELSE	GOTO	OPEN	RETURN
BASE 1	END	IF/THEN	PARAM	RUN
BYE	ENDEXIT	INPUT	PAUSE	SEEK
CHAIN	ENDIF	KILL	POKE	SHELL
CHD	ENDLOOP	LET	PRINT	STOP
CHX	ENDWHILE	LOOP	PUT	TROFF
CLOSE	ERROR	NEXT	RAD	TRON
CREATE	EXITIF/THEN	ON ERROR GOTO	READ	TYPE
DATA	FOR/TO/STEP	ON/GOSUB	REM	UNTIL
DEG	GET	ON/GOTO	REPEAT	WHILE/DO
DELETE	GOSUB		RESTORE	WRITE
DIM				

TRANSCENDENTAL FUNCTIONS

ACS (x)	COS (x)	LOG10 (x)	SIN (x)
ASN (x)	EXP (x)	PI	TAN (x)
ATN (x)	LOG (x)		

NUMERIC FUNCTIONS

ABS (x)	LAND (m,n)	MOD (m,n)	SQ (x)
FIX (x)	LNOT (m,n)	RND (x)	SQR (x)
FLOAT (m)	LOR (m,n)	SGN (x)	SQRT (x)
INT (x)	LXOR (m,n)		

STRING FUNCTIONS

ASC (char$)	LEFT$ (str$,m)	RIGHT$ (str$)	TRIM$ (str$)
CHR$ (m)	LEN (str$)	STR$ (x)	VAL(str$)
DATE$	MID$ (str$,m,n)	SUBSTR (st1$,st2$)	

MISCELLANEOUS FUNCTIONS

ADDR (var)	FALSE	POS	TAB (m)
EOF (#path)	PEEK (addr)	SIZE (var)	TRUE
ERR	INKEY(#<path>)	FILSIZ(#<path>)	

OPERATOR PRECEDENCE

highest ->	NOT	-(neg)				
		**				
	*	/				
	+ .	-				
	>	<	<>	=	>=	<=
	AND					
lowest ->	OR	XOR				

end of appendix b

USER NOTES

BASIC09 ERROR CODES

10 - Unrecognized Symbol
11 - Excessive Verbage (too many keywords or symbols)
12 - Illegal Statement Construction
13 - I-code Overflow (need more workspace memory)
14 - Illegal Channel Reference (bad path number given)
15 - Illegal Mode (Read/Write/Update/Dir only)
16 - Illegal Number
17 - Illegal Prefix
18 - Illegal Operand
19 - Illegal Operator

20 - Illegal Record Field Name
21 - Illegal Dimension
22 - Illegal Literal
23 - Illegal Relational
24 - Illegal Type Suffix
25 - Too-Large Dimension
26 - Too-Large Line Number
27 - Missing Assignment Statement
28 - Missing Path Number
29 - Missing Comma

30 - Missing Dimension
31 - Missing DO Statement
32 - Memory Full (need more workspace memory)
33 - Missing GOTO
34 - Missing Left Parenthesis
35 - Missing Line Reference
36 - Missing Operand
37 - Missing Right Parenthesis
38 - Missing THEN statement
39 - Missing TO

40 - Missing Variable Reference
41 - No Ending Quote
42 - Too Many Subscripts
43 - Unknown Procedure
44 - Multiply-Defined Procedure
45 - Divide by Zero
46 - Operand Type Mismatch
47 - String Stack Overflow
48 - Unimplemented Routine
49 - Undefined Variable

50 - Floating Overflow
51 - Line with Compiler Error
52 - Value out of Range for Destination
53 - Subroutine Stack Overflow
54 - Subroutine Stack Underflow
55 - Subscript out of Range
56 - Parameter Error
57 - System Stack Overflow
58 - I/O Type Mismatch
59 - I/O Numeric Input Format Bad

60 - I/O Conversion: Number out of Range
61 - Illegal Input Format
62 - I/O Format Repeat Error
63 - I/O Format Syntax Error
64 - Illegal Path Number
65 - Wrong Number of Subscripts
66 - Non-Record-Type Operand
67 - Illegal Argument
68 - Illegal Control Structure
69 - Unmatched Control Structure

70 - Illegal FOR Variable
71 - Illegal Expression Type
72 - Illegal Declarative Statement
73 - Array Size Overflow
74 - Undefined Line Number
75 - Multiply-Defined Line Number
76 - Multiply-Defined Variable
77 - Illegal Input Variable
78 - Seek Out of Range
79 - Missing Data Statement
80 - Print Buffer Overflow

Error codes above 80 are those used by OS-9 or other external programs. Consult the "OS-9 User's Guide" for a list of error codes and explanations.

end of appendix c

FUNCTION: Runb is the BASIC09 run-time package. It is similar to BASIC09 with tne following exceptions: Runb is about half the size of BASIC09 and no rile editing or debugging can be done. The main purpose of Runb is to save space and to execute packed modules. It should be noted that Runb will only execute packed modules. Another feature of Runb is that **CONTROL-C** and **CONTROL-E** can be trapped by **ON ERROR GOTO** where BASIC09 can't.

EXECUTION: When the name of a packed module is typed at the OS-9 prompt, Shell will determine that the module is packed BASIC09 I-code. Shell then loads and forks Runb, and Runb will link to and execute the named program. To run packed modules in this way, Runb must be in the commands directory.

NOTE: Packed modules can be executed without Runb if they are still within the workspace, but BASIC09 will have to be used and more space will be required.

end of appendix d

USER NOTES

INDEX

INDEX

INDEX

INDEX

INDEX