BASIC09
PROGRAMMING LANGUAGE
REFERENCE MANUAL

Revision H, January 1984

Microware System Corporation
P.O. Box 4865
Des Moines, Iowa 50304 U.S.A.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

INTRODUCTION

BASIC09 is an enhanced and structured BASIC language programming system specially created for the 6809 Advanced Microprocessor. In addition to the standard BASIC language statements and functions, BASIC09 includes many of the most useful elements of the PASCAL programming language so that programs can be modular, well-structured and use sophisticated data structures. It also 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. These features make BASIC09 an ideal language for many applications: scientific, business, industrial control, education, and more.

BASIC09 is unusual in that it is an Interactive Compiler that has the best of both kinds of language system: it gives the fast execution speed typical of compiler languages plus the ease of use and memory space efficiency typical of interpreter languages. BASIC09 is truly a complete PROGRAMMING SYSTEM that includes a powerful text editor, multipass compiler, run-time interpreter, 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 that provides all the tools and helpful "extra" facilities needed for fast, accurate creation and testing of structured programs.

BASIC09 FEATURES

* Structured, Recursive BASIC with PASCAL-type Enhancements
  - Allows Multiple, Independent, Named, Procedures
  - Procedure Call 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 Compiler Command Compacts Program and Provides Security
  - PRINT USING with FORTRAN-like Format Specifications
* 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
* 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
* 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
* High-Speed, High-Accuracy Math

- 9-Decimal-Digit 40-Bit-Binary Floating Point
- Full Set of Transcendentals (SIN, ASN, ACS, LOG, etc.)

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 development 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 of sophistication and reliability BASIC09 has today.

AN INTRODUCTION TO BASIC09

This section is intended for persons who have not previously written computer programs. If you are familiar with programming in general or BASIC programming specifically, this section can give you a "feel" for the BASIC09 interactive environment.

WHAT IS A PROGRAM?

A computer works something like a pocket calculator. With a calculator, you push a button, some calculation occurs, and the result is displayed. On some calculators you can write a program which is just a list of the buttons you want pushed, in the order you want them pushed, which is very similar to a computer program, but most computer languages use command names instead of buttons.

To get results from a computer, you must first put into the computer the list of commands you want executed in the order you want them executed. Each command will mean "do this thing" or "do that thing", but the computer only has certain commands which it will understand. A computer can do things like "add" or "save the result into a memory". Typing "get me a taco" to a computer won't get it; similarly, on a calculator you can't push buttons which aren't there. After you have stored a list of commands into the computer, you can tell it to perform those operations. This is like actually pushing the buttons on a hand calculator. Then, if you remembered to have the computer display your results, you get to see them. Generally, a computer does not automatically display results like a hand calculator. More calculations occur in a computer than in a calculator, and displaying all these results would simply be overwhelming.

You enter a program into a computer by using the computer itself as a "text editor" to store the commands you type in. Some editors allow you to enter any text you want. Other editors will only store valid computer commands. Even if the computer does store all the text you type in, it can only execute those commands it knows. If, during program execution, BASIC09 finds a word which does not correspond to a command, it will probably stop and print out an "error message". Other editors check each command as you enter it (usually after the carriage-return ending each line) and print error messages immediately for invalid commands. After typing in your list of commands, there are ways to display that list, to modify the commands you have typed in, and to insert others. But simply entering a computer program does not get results any more than thinking which buttons to push will get results on a calculator. You store your program by typing it into a computer, but no results are available until after you start the program running.

Even though programming is conceptually simple, it is easy to misspell commands which BASIC09 will not interpret correctly. Unlike humans, BASIC09 does not infer anything: Every command must be perfectly spelled and punctuated or it is wrong. Even after spelling errors are eliminated, it is likely that the sequence of commands you have entered will not do the job you wanted it to do. The meaning of the program to BASIC09 is often quite different than was intended by the programmer, but good intentions just don't push the right buttons. After you get the program to run without obvious error, you must test the program with sample input and see that it produces results which are known to be correct. If the results are incorrect, the program must be modified and tested until it does produce correct results. This process is known as testing and debugging. Computer malfunctions are rare, and if the computer works to store the program, it is probably working perfectly. If the program does not work, you need to puzzle out how the computer is doing something which you didn't realize that you told it to do. Programming can be frustrating, but if you enter the right commands, the computer will do the right things for you.

A SIMPLE BASIC09 PROGRAM

Probably the easiest way to explain programming is by example. This simple program sometimes keeps kids happy for hours. First, the program asks the user for his name. Then the computer types out "Hi", then the name, then "see you later". This may not seem like much, but it is great fun to type in things which are not your name, and see if they will be printed out. They will, of course.

When you turn on the BASIC09 computer it will print some heading information. If the prompt is "OS9: ", enter "basic09" (and a carriage-return) to get to the prompt "B:". When you have the prompt "B:", it means that the system is in the BASIC09 "command mode". While in the command mode, you can do several things like: list, kill, or create programs (called "procedures" in BASIC09). BASIC09 lets you keep several different programs in memory at the same time. Each program is identified by a name you give it when you create the procedure.

To create a new procedure you command the system to enter the "edit mode" by typing a simple "e" (in upper or lower case) and a carriage-return (the ENTER or RETURN key). The Editor lets you enter or change programs and actually checks for many common errors as you type in your program. Automatic checking feature is one of the nicest things about BASIC09. Because it's always "looking over your shoulder" to catch mistakes, it saves a lot of debugging time! If you're not 100% sure about how something works, you can go ahead and try it instead of digging though this manual. If you guess wrong, BASIC09 will usually show you where and why.

Because you did not specify a particular procedure name, BASIC09 will automatically select the name "PROGRAM" for you and will respond by printing out "PROCEDURE PROGRAM"; this means that you will be editing a procedure which is named PROGRAM. Later you will see that you can enter many different procedures and give them different names (just type the name you want to use for the program after the "e"). A procedure name may be any combination of alphanumeric characters beginning with a letter.

The computer output so far is as follows:

```
OS9: basic09
BASIC09
READY
B:e
PROCEDURE PROGRAM
*
E:
```

The asterisk (*) indicates the "current edit line" in the procedure being edited. In this case, the current line is empty since you have not yet entered anything. The asterisk is handy, since you will be moving back and forth between different lines to edit them. Later, you will be "opening" existing procedures for modification, and the first line will be displayed automatically, helping identify that you are editing the correct program.

When BASIC09 responds with the edit prompt "E:", it is in the edit mode. Now you can enter "edit commands" which help enter the computer program. While in edit mode, BASIC09 ALWAYS TAKES THE FIRST CHARACTER OF EVERY LINE AS AN EDIT COMMAND. Some of the basic edit commands are:

```
<space> <program statement> <cr>  insert a line
+ <cr>  go to next line down (just <cr> also does the same)
- <cr>  move back to previous line
L <cr>  list current line
d <cr>  delete current line
```

You must type an edit command at the start of each line. If you forget to type an edit command, BASIC09 will respond with "WHAT?". The most important edit command is the (invisible) space character; it means "save the following line of text". The "space" command is the way most text is entered into the system. If a line is to be entered, you must type a space before the rest of the line. Another useful edit command is "L*" (or "l*", since the editor accepts either upper or lower case) which will display the whole procedure. This allows you to watch the procedure develop as lines are entered.

You use the "space" command to enter the following line:

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

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

As mentioned before, BASIC09 checks for errors at the end of each line and again when the edit is finished. These errors are, in general, anything BASIC09 cannot identify or things that don't conform to the rules of the language. An error could be a bad character, mismatched parenthesis, or one of many other things. BASIC09 will print out an "error code" to identify the error and print an up arrow character under the place in the line where it detected the error. The error codes are listed at the end of this manual. If the error was detected at the end of the edit session, the I-code location of the error will also be printed. Cryptic information is all BASIC09 knows about the problem. Hopefully, it will help you to find and fix the error.

In the same way that you entered the first line, enter the following lines. Remember that the first character entered must be a space to get BASIC09 to save the rest of the line. Example:

```
E: INPUT name$
*
E: PRINT "Hi ";name$;", see you later."
*
E: END
*
```

The second line ("input name$"), when executed, commands BASIC09 to wait for a line of text to come in from the keyboard (this will happen after the user reads the message printed out in the first line). BASIC09 will accumulate text from the keyboard character-by-character until a carriage-return ends the line. This text is placed in the memory area corresponding to the variable "name$". The dollar-sign ($) on the end of the variable tells BASIC09 that you want to store a sequence of characters as opposed to a number.

The third line of procedure PROGRAM (print "Hi ";name$;", see you later."), starts out like the first line. The command "print" causes BASIC09 to print out the various values which come after it. When this line is executed, the characters H, i, and "space" are printed out since they are enclosed in double-quotes. Next, without additional spaces, BASIC09 prints out the line which was typed in by the user and saved in the memory corresponding to "name$" and prints out " see you later". When a PRINT statement contains multiple values, it will print them out one after the other. If the separator is a comma, BASIC09 will move to the next 16-column "tab stop" before printing the next value. However, if the separator between print values is a semicolon, absolutely no space will separate the values. The last line of the procedure ("END") tells BASIC09 to stop executing the program and to return to the command mode (B:). You have not yet EXECUTED the procedure, you are just EDITING. If you type in l*, the whole program will be listed as follows:

```
E:l*
```

```
PROCEDURE PROGRAM
 0000      PRINT "type your name"
 0012      INPUT name$
 0017      PRINT "Hi "; name$; ", see you later."
 0035      END
*
E:
```

Notice that the editor has added some information which you did not type in. You can use this listing to see exactly what to type in to run this program, but the editor only wants the relevant information.

The numbers to the left are "I-code addresses". These are the actual memory locations where each line begins relative to the start of the procedure. These number may look strange because they are in hexadecimal (base 16). These values are important, since the compiler may find errors at some I-code location and will try to convey what information it has to the programmer. I-code addresses are supplied automatically by BASIC09.

The space between the "I-code addresses" and the beginning of the program line is reserved for "line numbers". Line numbers are required in many versions of BASIC (although not in BASIC09). Notice that although the program was typed in lower case some words are printed in upper case. BASIC09 identifies valid command "keywords" and converts them to upper case automatically.

Now let's run it. First type "q" to quit the editor. We are now back in "command mode" (B:). Now type "run". BASIC09 remembers the last procedure edited (PROGRAM) and starts to execute it.

```
E:q
READY
B:RUN
type your name
? tex
Hi tex, see you later.
READY
B:
```

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

This program is extremely simple, but younger kids can get great fun from it. Its action is especially amusing to young people who are learning a computer language for the first time because a machine is "responding" to them, and because the machine is too easily "fooled" if you do not type in a real name.

BASIC PROGRAMMING TECHNIQUES: LOOPS AND ARITHMETIC

Another simple program that most of us can identify with is a program to print out multiplication tables.

```
PROCEDURE multable
  FOR i=1 TO 9
    FOR j=1 TO 9
      PRINT i*j; TAB(5*j);
    NEXT j
    PRINT
  NEXT i
```

First, open the editor by typing "e multable" as follows:

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

Next, type in the program line-by-line starting with "FOR i=1 TO 9" (lower-case is perfectly fine). If you loose your way, type "L*" to see where you are. The entire procedure will be displayed and an asterisk placed at the left of the current line. If you make a mistake, use "+" or "-" to move to that line, use "d" to delete the line, and use the space command to enter the line over. Make sure that there are no errors and then type "q". When you have the program running, try adding a statement before "FOR i=1 TO 9" as follows: "DIM i,j:INTEGER".

The FOR i=1 TO 9 and NEXT i constitute the start and end of a control structure or "loop". A control structure is used to cause repeated or conditional execution of the statement(s) it surrounds. A control structure usually has one entry at the top and one exit at the bottom. In this way, the entire structure takes on the properties of a single statement. The beginning statement of the FOR...NEXT structure (FOR...) provides "loop initialization", places the value 1 in the storage called "i", and sets up the operation of the following NEXT (every FOR must have a NEXT). When "NEXT i" is executed, the value in "i" is increased by 1 (which is the default STEP size) and compared to the value 9 (which is the ending value for this loop). If the resulting "i" is less than or equal to 9, the statement(s) following that FOR... is (are) executed.

Loops can be "nested" to execute the enclosed statements even more times. For example, the PRINT statement in "multable" is executed 81 times; once for each of 9 values of "j", and this number (9 times) for each of 9 values of "i". The ability to tremendously increase the number of times some code is executed is at the heart of both computer programming and programming errors. It means that a very small portion of a program can often be made to do the vast majority of the work. But a few remaining special cases may require individual handling and may consume more programming and code than that which "usually" works. Unfortunately, "usually" is not sufficient. A special case which occurs once in a thousand times may occur once a second, and if the error stops the program, further processing of normal values also stops. Experience has indicated that the programmer should know what is happening in the first and second pass, and the next-to-the-last and last pass through each loop in the program.

LISTING PROCEDURE NAMES

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

REQUESTING MORE MEMORY

BASIC09 automatically gets 4K bytes of workspace memory from OS-9 when it starts up. There is almost always more than this available, but BASIC09 does not grab it all so other tasks running on your computer can have memory too. If you are not multitasking and need more memory, the MEM command can get it if available. Just type MEM and the amount of memory you want. Depending on your computer and how it is configured, you can usually get at least 24K in OS-9 Level One Systems or 40K in OS-9 Level Two systems. For example:

```
MEM 20000
```

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 MEM responds with "WHAT?", the requested amount of memory is not available. There is another convenient way to request more memory when you first call up BASIC09 from OS-9. OS-9 has a "#" memory size option on the command line that lets you specify how much memory to give the program. To call BASIC09 with 16K of memory to start with, you can type:

```
OS9: basic09 #16K
```

STORING AND RECALLING PROGRAMS

Nobody wants to retype a whole program every time it is to be run. Two commands, **SAVE** and **LOAD**, are used to store programs and recall previously "SAVEd" programs to or from OS-9 disk files. The simplest way to use SAVE is by itself. It will store the procedure last edited or run on a disk file having the same name. For example:

```
B: SAVE
```

If our procedure name is the default name "PROGRAM", BASIC09 will create a file called "PROGRAM" to hold it. OS-9 won't let you have two files of the same name, because unique names are necessary to identify the specific file you want. Therefore, if a file called "PROGRAM" already exists, BASIC09 will ask you:

```
Overwrite?
```

If you respond "Y" for YES, it will replace the program previously stored in that file with the program to be saved, which is OK if what you want to save is a newer version of the same program, if not, you will permanently erase another program you may have wanted to keep. If this is the case, answer "N" for NO. Fortunately, there is a simple way to store the new procedure in a file using a different name: just type SAVE, a ">", and a different file name of your choice. The file name can consist of any combination of up to thirty-one letters, numbers, periods, or underscores ("_"). The only restriction is that the name must start with a letter A-Z or a-z. For example:

```
SAVE >newprogram5
```

will save the program in a file called "newprogram5". There are several useful variations of the **SAVE** command that let you save various combinations of programs in the same file. See the **SAVE** command description for more information. You should also read Chapter 2 of the "OS-9 Users Manual" to learn about the OS-9 commands that deal with disk files.

If you exit from BASIC09, it WILL NOT automatically save your programs. You must make sure to save them before you quit or they will be lost, unless they were saved at some time before!

The **LOAD** command, as its name implies, reads in a previously saved program from a disk file. You must give the name of the file with the command. For example:

```
LOAD program
```

If you just started BASIC09 and have not created any new procedures, this command is very straightforward. As the procedures stored in the file are loaded, BASIC09 displays their names as they are brought in. Once the program is loaded, you can edit and/or run it. But if you have a procedure in BASIC09 that has the same name as a procedure stored in the file, BASIC09 will replace it with the version loaded from the file. If this kind of conflict exists you could lose your original procedure, so be sure to save or RENAME it before loading another one (remember that BASIC09 can keep several procedures in memory at the same time as long as they have different names). If you want to permanently erase all other procedures before loading new ones, you can type:

```
B: KILL*
```

This tells BASIC09 to "kill" all procedures in memory and has the same effect as completely resetting BASIC09.

HOW TO PRINT PROGRAM LISTINGS

If your computer is equipped with a printer, you will want to make hard-copy listings of your programs. This is easy to do - just type:

```
B: LIST* /p
```

This tells BASIC09 to LIST all procedures in memory to the output device "/p" which is the printer device name in most OS-9 systems. Like the SAVE command, LIST has several useful variations. If you want to list just one procedure (and there is more than one in memory) you can type:

```
B: LIST procedurename >/P
```

If you want, you can list multiple procedures by replacing the single procedure name with a list of procedure names. Separate each procedure name from the next with a "space". An example is:

```
B: LIST procedurename1 procedurename2 procedurename3 >/P
```

Notice that if you omit the "/p" or ">/p" from the commands above, the program will be listed on your display instead of the printer. This is the same as the "L*" command in Edit Mode. You will also notice that the listing will be automatically "pretty-printed", e.g., program levels within loops are indented for easy reading.

BASIC09'S FOUR MODES

At any given time, BASIC09 is in one of four modes:

```
SYSTEM MODE:     executes system-oriented commands
EDIT MODE:       creates or changes procedures
EXECUTION MODE:  runs programs
DEBUG MODE:      used to test and verify programs
```

So far, you have been exposed to System Mode (SAVE, LOAD, etc.), Edit Mode (the editor), and Execution Mode (RUN). A section of this manual is devoted to each mode. The chart below shows how various commands in each mode will cause a change to another mode.

BASIC09 MODE CHANGE POSSIBILITIES

```
  OS-9          SYSTEM MODE           EDIT MODE
---------       ------------         ------------
        |       |          |         | +        |
        |       |  $       |         | -        |
        | <-----+--<eof>   |         | <cr>     |
        | <-----+--BYE     |         | <line#>  |
        |       |  CHD     |         | <space>  |
        |       |  CHX     |         | c        |
        |       |  DIR     |         | d        |
        |       |  EDIT----+-------> | l        |
        |       |  KILL    | <-------+-q        |
        |       |  LIST    |         | r        |       ------------
BASIC09-+-----> |  LOAD    |         | s        |       | TRON     |
        |       |  MEM     |         ------------       | TROFF    |
        |       |  PACK    | <--------------------------+ END or Q |
        |       |  RENAME  |                            | DEG/RAD  |
        |       |  RUN-----+-------> ------------       | STATE    |
        |       |  SAVE    | <-------+-END      |       | $        |
        |       |          | <-------+-<CTRL Q> |       | BREAK    |
        |       ------------ <-------+-STOP     | <-----+-CONT     |
BASIC09 |                            | PAUSE----+-----> | DIR      |
AUTORUN-+--------------------------> | ERROR----+-----> | LET      |
        |                            | <CTRL C>-+-----> | LIST     |
        | <--------------------------+-BYE      |       | PRINT    |
        |                            | PROGRAM  | <-----+-STEP     |
---------                            ------------       ------------
                                    EXECUTION MODE       DEBUG MODE
```

MORE ABOUT THE WORKSPACE...

The workspace concept is important because BASIC09 and OS-9 are both highly modular systems, and the workspace is a way to logically group a set of procedures (i.e. modules) which are applicable to a particular line of study or development. Modular software development lets the programmer divide a large and complex project into smaller, more manageable, and individually testable sections. Modularity also lets programmers accumulate and use libraries of commonly used routines.

As the software is written and debugged, BASIC09 makes it easy to deal with the procedures that comprise an overall project, either individually or as a group. For example, you can save all procedures in the workspace to a single mass storage file or load a file containing multiple procedures. Usually all procedures associated with a project exist inside the workspace. However, you can also call library procedures which are "outside" the workspace in OS-9 memory module format. The library procedures can be written in BASIC09 or machine language, can be in RAM or ROM memory, and can even be shared by several users.

BASIC09 always reserves approximately 1.2K bytes of the workspace for internal use. All remaining space is used for storage of procedures and for procedure variable storage during execution. BASIC09 will not run a procedure if there is not enough space for variables. If you run out of workspace area, you can use the MEM command to enlarge the workspace or you can kill procedures in the workspace that are not needed. The "MEM" command can be used at any time to change the size of the workspace. The size of the workspace can be increased (subject to availability of free memory) or decreased (but not below the minimal amount needed to store the present contests of the workspace).

WHERE TO GO FROM HERE?

A good way to learn BASIC09 is to use it! Try typing in and running some of the example programs in the back of the book. Look up and study the function of each program statement. Read the chapters on the EDIT and DEBUG modes and experiment with more advanced commands. Since BASIC09 and the OS-9 Operating System are so intimately connected, a basic understanding of OS-9 is necessary. See Chapter 2 of the "OS-9 OPERATING SYSTEM USER'S MANUAL.

This Page Intentionally Blank

SYSTEM MODE COMMANDS

System Mode includes commands to save, load, 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

$	CHX	EDIT	LOAD	RENAME
BYE	DIR	KILL	MEM	RUN
CHD	E	LIST	PACK	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 don't operate on specific procedures but may have optional or required parameters. Other 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. For example:

```
LIST pete
```

will display the procedure named "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. For example:

```
LIST*
```

will display all of the procedures in the workspace.

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 that require a file name as well as (one or more) procedure names 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. 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
```

or

```
SAVE tic,tac,toe
```

which is exactly equivalent to

```
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 which enters Execution Mode to run a procedure

EDIT which enters Edit Mode to create or change a procedure

The one other mode, Debug Mode, cannot be entered directly from the system mode - more on this later.

SYNTAX NOTATION USED IN DESCRIPTIONS OF SYSTEM COMMANDS

Individual descriptions of each system command follow. In order to precisely describe their formats, the syntax notation shown below is used.

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

$ [<text>] ("Shell" Command)

This command calls the OS-9 Shell command interpreter to process an OS-9 command or to run another program. Running the OS-9 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: $asm sourcefile&	calls the assembler as a background task
B: $basic09 fourier(20)&	starts another concurrent BASIC09 program

BYE (or ESCAPE character)

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 escape key (technically speaking, an end-of-file condition on BASIC09's standard input path) does the same thing.

CHD <pathlist> or CHX <pathlist>

CHD changes the current OS-9 user Data or Execution Directory to the specified pathlist which must be a directory file. BASIC09 uses the Data Directory to LOAD or SAVE procedures. The Execution Directory is used to PACK or auto-load packed modules. An example follows.

Example:

```
CHD /d1/joe/games
```

DIR [<pathlist>]

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 [<procname>]
E [<procname>]

EDIT (E) exits command mode and enters the text editor/compiler mode. If the procedure named does not exist, a new one is created.

See Chapter 4 for a complete discussion of how edit mode works.

Examples:

```
E newprog

EDIT printreport
```

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

KILL erases the procedure(s) specified. KILL* clears the entire workspace. The process may take some time if there are many procedures in the workspace.

Examples:

```
KILL formulas

KILL prog1,prog3,prog7
```

```
LIST  [<procname> {,<procname>}] [> <pathlist>]
LIST* [<pathlist>]
```

LIST prints a formatted "pretty printed" listing of one or more procedures. The listing includes the relative I-code storage addresses in hexadecimal numbers 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.

The **LIST**, **SAVE**, and **PACK** commands all have identical syntax, except that **LIST** prints on the OS-9 Standard Error Path (#2) if no pathlist is given. The files produced are formatted differently, but the function is similar.

IMPORTANT NOTE: If an "*" is used with **LIST**, **SAVE**, or **PACK**, the file name follows immediately WITHOUT a ">" before it!

Examples:

```
LIST* /p

LIST prog2,prog3 >/p

LIST prog5 >temp
```

```
LOAD <pathlist>
```

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
MEM <number>

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 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 [<procname> {,<procname>}] [> <pathlist>]
PACK* [<pathlist>]

PACK causes an extra compiler pass on the procedure(s) specified which removes names, line numbers, non-executable statements, etc. The result is a smaller, faster procedure(s) that CANNOT 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 a default pathname. 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. THE RESULTING FILE CANNOT BE LOADED INTO THE WORKSPACE LATER ON, so you should always perform a regular **SAVE** before **PACK**ing a procedure!

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 (kill) 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. Examples follow.

Examples:

    PACK procl,proc2 >packed.programs

    PACK* packedfile

RENAME <procname>,<new procname>

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

Example:

    RENAME thisproc thatproc

RUN [<procname> [(<expr> , {<expr>})]]

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

A parameter list can be used to pass expected parameters to the procedure in the same way a RUN statement inside a procedure calls another procedure except for the restriction that all parameters must be constants or expressions without variables. See the PARAM statement description. Assembly language procedures cannot be run from System Mode.

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 [<procname> {,<procname>} [> <pathlist>]]
SAVE* [<pathlist>]

Writes the procedure(s) (or all procedures) 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.

Examples:

```
SAVE proc2,proc3,proc4 >monday.work

SAVE* newprogram

SAVE

SAVE  >testprogram
```

Edit Mode (also called "The Editor") is used to enter or modify BASIC09 procedures. It is entered from System Mode by the EDIT (or E) command. As soon as Edit Mode is entered, prompts change from "B:" to "E:". 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.

OVERVIEW OF EDIT COMMANDS

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

EDIT MODE COMMANDS

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

HOW THE EDITOR WORKS

In order to understand how the editor works it is helpful to have a general idea of what goes on inside BASIC09 while you are editing procedures. 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 computers 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. It is this innovative system that gives BASIC09 its most powerful and unusual abilities.

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 <u>always</u> 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 #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

As mentioned previously, the editor has the capability to work on programs with or without line numbers (or both). 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:

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

To enter or replace a numbered line, simply type in the line number and statement. Numbered lines can be entered in any order but will be automatically stored in ascending sequence. To move to a numbered line, type the line number followed by a carriage return. The editor will move to that line (or the one with the next higher number if not found) and print 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. Its formats are:

```
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 an initial line number, and each <incr> is added to the previous line number for the next line's number. For example,

```
r 200,5
```

will give the first line number 200, the second 205, the third 210, etc. If <beg line#> and/or <incr> are not specified, the values 100 and 10, respectively, are assumed. 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 has to be another way to tell BASIC09 what place in the program to work on. To do this, the editor maintains an "edit pointer" that indicates which line is the present working location within the procedure, and commands start working at 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 n lines
-*	moves to the beginning of the procedure
+	moves forward one line
+ <number>	moves forward N lined
+*	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 the "space" character 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. The second form deletes ALL lines in the procedure (caution!). The editor accepts "+*" and "-*" to mean to the end, or to the beginning of the procedure respectively. 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.

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

What's a string? A string is a sequence of one, two, or more characters that 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 punctuation characters (called <u>delimiters</u>) so the editor knows where the string begins and ends. The characters used for delimiters are not considered part of the string and cannot 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 different creatures.

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 on the current edit pointer line onward. 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 cannot be located, the message:

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

will be displayed and the edit pointer will remain at its original position. 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 ends up at 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 "c" 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 very commonly used for things like: fixing lines with errors without having to retype the entire line, changing a variable name throughout a program, etc. Its formats are:

```
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 causes a line with an error. It cannot be used to find or change line numbers.

A word of warning: sometimes you can inadvertently change a line you didn't 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 follow.

Examples:

```
c/xval/yval/  and  c*,GOSUB 5300,GOSUB 5500
```

This Page Intentionally Blank

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 type RUN without giving a procedure name at all (the "*" shown in the DIR command identifies this procedure).

If the procedure expects parameters (see Chapter 7), they can be given on the same command line, however they must all be constant numbers or strings, as appropriate, and must be given in the correct order. For example:

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

is used to call a program that expects parameters, such as

```
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-Q

3. A run-time error occurs

4. You type CONTROL-C (<SHIFT><BREAK>)

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 don't 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 a number of 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. For example, the following OS-9 command lines use this feature:

```
OS9: BASIC09 printreport("Past Due Accounts")

OS9: BASIC09 evaluate(COS(7.8814)/12.075,-22.5,129.055)
```

OVERVIEW OF DEBUG MODE

One of BASIC09's outstanding features is its set of powerful symbolic debugging commands. What is Symbolic Debugging? Simply stated, it is testing and manipulation of programs using the actual names and program statements used in the program. In this chapter you will learn how Debug Mode can let you watch your program run in slow motion so you can observe each statement as it is executed. As a bonus, you will also learn how to use 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

$ <text> (Shell Command)

Calls OS-9's Shell command interpreter to run a program or OS-9 command. Exactly the same as System Mode "$" command.

BREAK <proc name>

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. You can now enter:

```
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't put a breakpoint on a procedure unless it has been called but not yet returned to. Hence, BREAK cannot be used on procedures that have not yet been run.

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 either degrees or radians as the angle unit measure used by trigonometric functions. These commands only affect the procedure currently being debugged or run.

DIR [<pathname>]

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

Q

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

LET <var> := <expr>

Essentially the same as the BASIC09 LET program statement, which allows the value of a procedure variable to be set to a new value using the result of evaluation of the 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 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 [#<expr>,] [USING <expr>,] <expr list>

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, and no new variable names can be used. User-defined data structures can not be printed.

STATE

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 the suspended procedure. An example:

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

STEP [<number>] or <CR>

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, so the original source lines can 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

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 (unless of course, other called procedures have trace mode on).

DEBUGGING TECHNIQUES

If your program doesn't do what you expect it to, it is bound to show one of two symptoms: incorrect results, or premature termination due to an error. The second case will automatically send you into Debug Mode. In the first 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're 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 it where you think things begin to go amiss, and then you rerun the program. When the program hits the **PAUSE** statement and enters DEBUG mode, it is time to turn the trace mode on and actually watch your program run. To do so, just type:

```
D: TRON
```

After you have done this, you hit the carriage return key once for every statement. 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. But what if in the process of doing so you encounter a loop that works OK but executes 200 statements repetitively? That's a lot of carriage returns. 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 example is:

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

Don't forget that trace mode is "local" to one procedure only. If the procedure under test returns to another procedure you will need to use the **BREAK** command or a 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: it 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't 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:
```

Don't forget that the Debug Mode **PRINT** command can use **PRINT USING** to produce formatted output (including hexadecimal).

By adding less than a dozen statements to the program, you can make it store its variables on a disk file so they're remembered from session to session. There are also many other enhancement possibilities.

WHY ARE THERE DIFFERENT 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 kinds of data. Some data are numbers used for counting or measuring. Another example is textual data composed of letters, punctuation, etc., such as your name. Seldom can they be mixed (for example multiplication is meaningless to anything but numbers), and they have different storage size requirements. Even within the same general kind of data, it is frequently advantageous to have different ways to represent data. For example, 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. A good analogy is to consider the five basic types to be atoms, and the new types you create as molecules. This is why the five basic types are called <u>atomic data types</u>.

DATA STRUCTURES

A <u>data structure</u> refers to storage for more than one data item under a single name. 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 <u>array</u>, 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. The ":INTEGER" part 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
```

As you shall see, BASIC09 lets you create complex arrays and even arrays that have different data types combined.

ATOMIC DATA TYPES

BASIC09 includes five atomic 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:

BASIC09 ATOMIC DATA TYPE SUMMARY

Type	Allowable Values	Memory Requirement
BYTE	Whole Numbers 0 to 255	One byte
INTEGER	Whole Numbers -32768 to 32767	Two bytes
REAL	Floating Point +/- 1*10^38	Five Bytes
STRING	Letters, digits, punctuation	One byte per char.
BOOLEAN	True or False	One byte

Why are there three different ways to represent numbers? Although REAL numbers appear to be the most versatile because they have the greatest range and are floating-point, arithmetic operations involving them are relatively slow (by a factor of about four) compared to the INTEGER or BYTE types. Thus using INTEGER values for loop counters, indexing arrays, etc. can significantly speed up your programs. The BYTE type is not appreciably faster than INTEGER, but 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 use the REAL type.

Type BYTE

BYTE variables hold integer values in the range 0 through 255 (unsigned 8-bit data) which are stored as a single byte. BYTE values are always converted to another type (16-bit integer values and/or real values) for computation, thus they have no speed advantage over other numeric types. However, BYTE variables require only half the storage used by integers, and 1/5 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.

Type INTEGER

INTEGER variables consist of two bytes of storage and hold a numeric value in the range -32768 through 32767 as signed 16-bit data. Decimal points are not allowed. INTEGER constants may also be represented as hexadecimal values in the range $0000 through $FFFF 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 65536; i.e., 32767 + 1 yields - 32768. Division of an integer by another integer yields an integer result, and any remainder is discarded. The programmer should be aware that numeric comparisons made on values in the range 32767 through 65535 will actually be dealing with negative numbers, so it may be desirable to limit such comparisons to tests for equality or non-equality. Additionally, certain functions (LAND, LNOT, LOR, LXOR) use integer values but produce results on a non-numeric bit-by-bit basis.

Type REAL

The REAL 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 5 consecutive memory bytes. The first byte is the (8-bit) exponent in binary two's-complement representation. The next four bytes are the binary sign-and-magnitude representation of the mantissa; the mantissa in the first 31 bits, and the sign of the mantissa in the last (least-significant) bit of the last byte of the real quantity.

INTERNAL REPRESENTATION OF REAL NUMBERS

```
        +--------+--------+--------+--------+--------+
        |exponent|              mantissa           |S| <- mant. sign
        +--------+--------+--------+--------+--------+
  byte:     +0       +1       +2       +3       +4
```

The exponent covers the range 2.938735877 * 10^-39 (2^-128) through 1.701411835 * 10^38 (2^127) 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 0.5 through .9999999995 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 cannot 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.

Type STRING

A STRING is a variable-length sequence of characters or nil (an empty STRING). A variable may be defined as a STRING either explicitly (e.g., DIM title:STRING) or implicitly by appending the dollar-sign character to the identifier (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. Notice that 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 allotted to that 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.

```
        +--------+--------+--------+--------+--------+--------+
        |   S    |   A    |   M    |   00   |        |        |
        +--------+--------+--------+--------+--------+--------+
 byte:     +0       +1       +2       +3       +4       +5
```

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

```
        +--------+--------+--------+--------+--------+--------+
        |   R    |   O    |   B    |   E    |   R    |   T    |
        +--------+--------+--------+--------+--------+--------+
  byte:    +0       +1       +2       +3       +4       +5
```

Type BOOLEAN

A BOOLEAN quantity has only two values: TRUE or FALSE. A variable may be typed BOOLEAN (e.g., DIM done_flag:BOOLEAN). BOOLEAN quantities are stored as single byte values, but they may not be used for numeric computation. BOOLEAN values print out as the character strings: "TRUE" and "FALSE". BOOLEAN values result from comparisons (comparing two compatible types), and are appropriate for logical flags and expressions (result:=a AND b AND c). Do not confuse BOOLEAN operations AND, OR, XOR, and NOT (which operate on the BOOLEAN values TRUE and FALSE) with the logical functions LAND, LOR, LXOR, LNOT (which 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 run-time error.

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 cannot 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 even if the number could have been stored in INTEGER or BYTE format. Thus 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. Here are some examples of legal real constants:

```
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 -32768 to +32767 are treated as INTEGER numbers. BASIC09 will also accept integer constants as unsigned decimal numbers in the range 0 to 65535 or in hexadecimal in the range 0 to $FFFF. Hex numbers must have a leading dollar sign. Here are some examples of integer constants:

```
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 double quote characters. The binary value of each character byte can be 1 to 255. Double quote characters to be included in the string use two characters in a row to represent one double quote. The null string "" is important because it represents a string having no characters. It is analogous to the numeric zero. Here are some examples of string constants:

```
"BASIC09 is a new microcomputer language"
"AABBCCDD"
""   (a null string)
"An ""older man"" 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 either "by reference", allowing values to be returned from the called procedure, or "by value", which 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 parenthesis 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 don't 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 a new 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't 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 (two-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", while a 2 or 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. Thus, a matrix 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 atomic 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 atomic 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. Additionally, 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.

This Page Intentionally Blank

EVALUATION OF EXPRESSIONS

Many BASIC09 statements evaluate expressions. The result of an evaluation is just a value of some atomic type (e.g. 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 (e.g., 5.0 , 5 , "5" , or TRUE), a variable name, or a function (e.g., SIN(x)) 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 (as is often the case) 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 (assigned to) 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 "atomic" data types, i.e., real, integer, byte, boolean, or string. 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. Here's an example:

```
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 take two operands (except negation) and cause some operation to be performed producing 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. "NUMERIC" refers to either BYTE, INTEGER, or REAL types.

BASIC09 EXPRESSION OPERATORS

Operator	Function	Operand Type	Result Type
-	Negation	NUMERIC	NUMERIC
^ or **	Exponentiation	NUMERIC	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 (i.e., multiplications performed before addition). 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)` (returns 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 expressions
<str>	means any string-result expression
<int>	means any integer-result expression

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

SIN(<num>)	trigonometric sine of <num>
COS(<num>)	trigonometric cosine of <num>
TAN(<num>)	trigonometric tangent of <num>
ASN(<num>)	trigonometric arcsine of <num>
ACS(<num>)	trigonometric arcosine of <num>
ATN(<num>)	trigonometric arctangent of <num>
LOG(<num>)	natural logarithm (base e) of <num>
LOG10(<num>)	logarithm (base 10) of <num>
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>)	largest whole number less than or equal to <num>
PI	the constant 3.14159265
SQR(<num>)	square root of <num>, which must be positive
SQRT(<num>)	square root of <num>; same as **SQR**

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 can return ANY numeric type, depending on the type of the input parameter(s).

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 return results of type **INTEGER** or **BYTE**

FIX(<num>) round **REAL** <num> and convert to type **INTEGER**

MOD(<num1>,<num2>)
modulus (remainder) function, <num1> mod <num2>

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>

POS current character position of **PRINT** buffer

ASC(<str>) numeric value of first character of <str>

LEN(<str>) length of string <str>

SUBSTR(<str1>,<str2>)
substring search: returns starting position of first occurrence of <str1> in <str2>, or 0 if not found.

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.

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:

CHR$(<int>)	ASCII char. equivalent of <int>
DATE$	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:

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.

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. Generally, the programmer will create a main routine which will then 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, and more than one program statement can be put on the same line if separated by "\" (<clear></>) 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 when using BASIC09 since 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.

Program readability is improved if all variables are declared with DIM statements at the beginning of the procedure, but this is not mandatory. The program can be terminated with **END** or **STOP** statements, which are also optional.

LINE NUMBERS

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) because 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.

ASSIGNMENT STATEMENTS

Assignment statements are used for computation or initialization of variables.

LET Statement

```
Syntax: [LET] <var> := <expr>
        [LET] <var> = <expr>
        [LET] <struct> := <struct>
        [LET] <struct> = <struct>
```

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 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 used in PASCAL.

Another usage 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. In fact 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 <int expr> , <byte expr>

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.

Examples:

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

POKE 1200,14

POKE $1C00,$FF

POKE pointer,PEEK(pointer+1)

(* alternative to: 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 (usually) sequential execution of program statements. They are used to construct loops or make decisions that alter program flow. BASIC09 provides a selection of looping statement forms that allow any kind of loop to be constructed using sound structured programming style.

IF Statement: Type 1

Syntax: IF <bool expr> THEN <line #>

This form of the if statement causes execution to be transferred to the statement having the line number specified if the result of the expression is TRUE, otherwise the next sequential statement is executed. For Example:

```
IF payment < balance then 400
```

IF Statement: Type 2

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

This kind of IF structure evaluates the 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 otherwise to the statement following the ENDIF.

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
```

FOR/NEXT Statement

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

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> which 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. There is no limit to the nesting of FOR/NEXT loops.

Examples:

```
FOR counter = 1 to 100 STEP .5
  PRINT counter
NEXT counter


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
        ENDWHILE
```

This is a loop construct with the test 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
        UNTIL <bool expr>

This is a loop that has its test 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 and ENDLOOP Statements

Syntax: LOOP
ENDLOOP

EXITIF and ENDEXIT Statements

Syntax: EXITIF <bool expr> THEN <statements>
ENDEXIT

These related types of statements 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 that if there is no EXITIF clause you will create a loop that never ends.

The EXITIF clause evaluates an expression to a boolean result. If the result is false, the statement following the ENDEXIT is executed next. Otherwise, the statement(s) between the EXITIF and ENDEXIT are executed, then control is transferred to the statement following the body of the loop. 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 BASIC09 loop construct (e.g., FOR/NEXT, REPEAT... UNTIL, etc.). Examples:

```
LOOP
  count=count+1
EXITIF count >100 THEN
  done = TRUE
ENDEXIT
  PRINT count
  x = count/2
ENDLOOP
```

is equivalent to

```
100 REM top of loop
    count=count+1
    IF COUNT <= 100 then 200
    done = TRUE
    GOTO 300
200 PRINT count
    x = count/2
    GOTO 100
300 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 #>

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 #>
        RETURN

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 <int expr> GOTO <line #> {,<line #>}
        ON <int expr> GOSUB <line #> {,<line #>}

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

The expression must evaluate to a positive INTEGER-type 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. If the expression has any other result, no step is selected and the next sequential statement is executed.

Example:

```
     (* spell out the digits 0 to 9 *)
     DIM digit:INTEGER
     A$="one digit only, please"
     INPUT "type in a digit"; digit
     ON digit+1 GOSUB  10,11,12,13,14,15,16,17,18,19
     PRINT A$
     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
  18 A$ := "EIGHT"
     RETURN
  19 A$ := "NINE"
     RETURN
```

ON ERROR GOTO Statement

Syntax: ON ERROR [GOTO <line #>]

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], 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
```

RUN Statement

Syntax: RUN <proc name> [(<param> {,<param>})]
    or: RUN <string var> [(<param> {,<param>})]

This statement calls a procedure by name; when that procedure ends, control will pass to the next statement after the RUN. It is most often used to call a procedure inside the workspace, but it can also be used to call a previously compiled (by the PACK command) procedure or a 6809 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, but BYTE arrays are O.K.). If a parameter is a constant or expression, it is passed "by value", i.e., it 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", i.e., 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't be found in the workspace, BASIC09 will check to see if it was loaded by OS-9 outside the workspace. If it isn't 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 6809 machine language module and executes it accordingly. If it is a 6809 machine language module, BASIC09 executes a JSR instruction to its entry point and the module is executed as 6809 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.

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

```
+----------------------+          ^
|                      |          |
                          higher addresses

|   more parameters    |

|                      |
+----------------------+         ---
|                      |          |
|  size of 1st param   |          |
+  - - - - - - - -     +      4 bytes
|  addr  of 1st param  |          |
|                      |          |
+----------------------+         ---
|                      |          |
|  parameter count     |      2 bytes
|                      |          |
+----------------------+         ---
|                      |          |
|  return address      |      2 bytes
|                      |          |
+----------------------+         ---   <- 6809 Stack Pointer
                                          Register value
```

Machine language modules return error status by setting the "C" bit of the MPU condition codes register and by setting the B register to the appropriate error code. For an example of a machine language subroutine ("INKEY"), see Appendix A.

Example of use of the RUN statement:

```
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 <str expr>

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.

WARNINGS:

1. It can be fatal to OS-9 to KILL a procedure that is still "active".

2. When used together 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 <str expr>

The **CHAIN** statement performs an OS-9 "chain" operation on the **SHELL**, passing the specified string as an argument. **CHAIN** causes BASIC09 to be exited, unlinked, and its memory to be returned to OS-9. The string 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 User's Guide for more details on the capabilities of the Shell.

Files that are open when a **CHAIN** occurs are not closed. However, the OS-9 Fork call will only pass the standard I/O paths (0,1,2) to a child process. Therefore, if it is necessary to pass an open path to another program segment, 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 <str expr>

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, and more. Consult the "OS-9 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 (subject to the normal security functions). The string expression is evaluated and passed to the shell to be executed as a command line (just as if it had been typed in). If the string is null, 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.

Here are a few examples of using the shell from BASIC09:

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

END Statement

Syntax: END [<output list>]

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 also works 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>]

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

BYE Statement

Syntax: BYE

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!

ERROR Statement

Syntax: ERROR(<int expr>)

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. For details on error handling see the ON ERROR GOTO statement description.

PAUSE Statement

Syntax: PAUSE [<output list>]

PAUSE suspends execution of the procedure and causes BASIC09 to enter Debug Mode. If an output list is given, it also works like a PRINT statement.

```
   <output>    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 <str expr>
        CHX <str expr>

These statements change the current default Data or Execution directories, respectively. The string must specify the pathlist of a file which has the DIR attribute. For more information on the OS-9 directory structure, consult the "OS-9 User's Manual".

DEG and RAD Statements

Syntax: DEG
        RAD

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.

BASE 0 and BASE 1 Statements

Syntax: BASE 0
        BASE 1

These statements indicate whether a particular procedure's lowest array or data structure index (subscript) is zero or one. The default is one. These statements do not affect the string operations (e.g., MID$, RIGHT$, OR LEFT$) where the beginning character of a string is always index one.

TRON and TROFF Statements

Syntax: TRON
        TROFF

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. Also, the result of each expression evaluation is printed as it occurs.

Comment Statements

```
Syntax: REM <chars>
        (* <chars> [ *] )
```

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
```

DECLARATIVE STATEMENTS

The **DIM**, **PARAM**, and **TYPE** statements are called _declarative 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 don't need to for simple variables of type **REAL**, because this is the default format for undeclared variables. You also don't need to for 32-character **STRING** type variables (any name ending with a "$" is automatically assigned this type). Even though you don't have to declare variables in these two cases, you may want to anyway to improve you program's internal documentation. Those 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 atomic data types built-in to BASIC09. Therefore, the **TYPE** statement is only used in programs that utilize complex data structures.

DIM Statement

```
Syntax DIM <decl seq> {; <decl seq> }
           <decl seq> := <decl> {, <decl> } [: <type> ]
           <decl> := <name> [ <subscript> ]
           <subscr> := ( <const> [,<const> [,<const>]] )
           <type> := BYTE | INTEGER | REAL | BOOLEAN |
                     STRING | STRING <max len> | <user defined type>
           <user def> := user defined by TYPE statement
```

The DIM statement is used to declare simple variables, arrays, or complex data structures of the five atomic 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 made type REAL or type STRING[32] if the name ends with a "$" character. Therefore all simple variables of other types must be explicitly declared. For example:

```
DIM logical:BOOLEAN
```

Several variables can be declared in sequence with a :<type> following a group of the same type:

```
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 (including type grouping) 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 32767. 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 matrix of integers, "c" is a simple integer variable, and "x" is a three-dimensional array of 12-character strings.

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

(* 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: Same as DIM statement

PARAM is 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 ensure that the total size of each parameter (as evaluated by the **RUN** statement in the calling procedure) 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 (to prevent accidental access to storage other than the parameter), but it DOES **NOT** CHECK **TYPE**. However, in most cases 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 really know what you are doing 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> {; <type decl>}
             <type decl>     := <field name> . <decl> [ : <type> ]
             <decl>          := <name> [ <subscript> ]
             <subscript>     := ( <const> [,<const>] {,<const>] ] )
             <type>          := BYTE | INTEGER | REAL | BOOLEAN |
                                STRING [<max len>] | <user defined>
             <user defined> := user defined by TYPE statement
```

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. Here's an 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 non-rectangular data 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.

This Page Intentionally Blank

FILES AND UNIFIED INPUT/OUTPUT

A file is a logical concept for a sequence of data which is 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, so 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 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 that 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 (**WRITE/READ**) textual data only in order. 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 an arbitrary number of characters terminated by a carriage return. Each **READ** will also read all characters up to the next carriage return.

A random-access file sends and receives (**PUT/GET**) data in binary form exactly as it is internally represented in BASIC09 which 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** of a structure 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 (using **SEEK**) and to begin to **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 terminated by CLOSE 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 from a given program outward, or from some device inward. In order for data to flow to or from a device, there must be in OS-9 an associated device driver - see the OS-9 Users Manual. 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. Three path numbers have special meanings because they are "standard I/O paths" representing BASIC09's interactive input/output (your terminal). 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 (Display)
2 Standard Error/Status (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 [#<int expr>,] ["<prompt>",] <input list>

INPUT accepts input during execution of a program. The input is normally read from the standard input device (terminal) unless an optional path number is given. When the INPUT statement is encountered, program execution is suspended and a "?" prompt is displayed. If the optional prompt string is given, it is displayed instead of the normal "?" prompt. The INPUT statement is really <u>both</u> an input and output statement. Therefore, if a path other than the default standard input path is used, the path should be open in UPDATE mode. This makes INPUT 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 atomic 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 cannot exceed 256 characters.

If any error occurs (type mismatch, insufficient amount of data, etc.), the message:

```
**INPUT ERROR - RETYPE**
```

is displayed, followed by a new prompt. The entire input line must then be reentered.

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 (i.e., to read pure binary data), use the GET statement.

Examples:

```
INPUT number,name$,location

INPUT #path,x,y,z

INPUT "What is your selection: ", choice

INPUT #path,"What's your name? ",name$;
```

Here's how to read a single character (without editing) from the terminal (path #0):

```
DIM char:STRING[1]
GET #0,char
```

For a function to test if data is available from the keyboard without "hanging" the program, see the "INKEY" assembly language program included in Appendix A.

PRINT Statement

```
Syntax: PRINT <output list>
        PRINT #<int exp>, <output list>
        PRINT USING <str expr>, <output list>
        PRINT #<int exp>, USING <str expr>, <output list>
```

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 comma or semicolon characters. Each item can be a constant, variable, or expression of any atomic 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 spacing in between. 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 long 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.

The "TAB(expr)" function can be used as an item in the output list, which outputs the correct number of spaces to cause the next item to start in the print column specified by the result of the expression. If the output line is already past the desired tab position, the TAB is ignored. A related function, "POS", can be used in the 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. A practical value is at least 512 characters.

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 USING "R10.2,X2,R5.3",x,y
```

```
PRINT #outpath USING fmt$,count,value

(* print an 80-character line of all dashes *)
REPEAT
  PRINT "-";
UNTIL POS >= 80
PRINT
```

OPEN Statement

```
Syntax: OPEN #<int var>,<str expr> [ : <access mode> ]
        <access mode> := <mode> ! <mode> + <access mode>
        <mode>        := READ ! WRITE ! UPDATE ! EXEC ! DIR
```

OPEN issues a request to OS-9 to open an I/O path to an existing file or device. The STRING expression is evaluated and passed to OS-9 as the descriptive pathlist. The variable name specified 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 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 with WRITE or UPDATE modes. The EXEC mode causes the current execution directory to be used instead of the current data directory. Refer to the "OS-9 User's Manual" for more information on how files access modes.

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 #<int var>,<str expr> [ : <access mode> ]
        <access mode> := <mode> ! <mode> + <access mode>
        <mode>        := WRITE ! UPDATE ! EXEC
```

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 multifile, this statement works like an "OPEN" statement. The variable name is used to "receive" the path number assigned by OS-9 and must be of BYTE or INTEGER type. The STRING expression is evaluated and passed to OS-9 to be used 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". File size may be set explicitly using the OS9 statement.

Examples:

```
CREATE #trans,"transactions":UPDATE

CREATE #spool,"/user4/report":WRITE

CREATE #outpath,name$:UPDATE+EXEC
```

CLOSE Statement

Syntax: CLOSE #<int expr> { ,#<int expr> }

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.

Paths #0, #1, and #2 (the standard I/O paths) 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 <str expr>

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 on the file is permanently lost. The string expression is evaluated and passed to OS-9 as the descriptive pathlist of the file.

The user must have write permission for the file to be deleted. See the "OS-9 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 #<int expr>,<real expr>

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 on files using the GET and PUT statements.

The first expression specifies the path number of the file and must evaluate to a byte value. The second expression specifies the desired file pointer address, and must evaluate to a REAL value in the range 0 <= result <= 2,147,483,648. 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
```

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)
```

WRITE Statement

Syntax: WRITE #<int expr>,<output list>

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 a **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
```

READ Statement

Syntax: READ #<int expr>,<input list>

The READ statement causes input data in ASCII character format to be read from a file or device. The first expression specifies a path number which 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.

GET Statement

PUT Statement

Syntax: GET #<expr>,<struct name>
        PUT #<expr>,<struct name>

The GET and PUT statements read and write fixed-size binary data records to files or devices. These are the primary I/O statements used for random access input and output.

The first expression 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. For example:

```
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
```

You can manually count the number of bytes assigned for each type to calculate the total size of each record. Sometimes this can become complicated and error-prone. Also, any change in a TYPE definition could require recalculation. Fortunately, BASIC09 has a built-in function:

```
SIZE(<name>)
```

that returns the number of bytes assigned to any variable, array, or complex data structure. In our example, SIZE(work_rec) will return the number 37, and SIZE(inv_array) will return 3700. 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 typically require fewer statements to perform I/O functions than in other BASICs.

4. It is versatile. By creating appropriate data structures you can read or write almost any kind of data on any file, including files created by other programs or languages.

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> ]
```

These statements provide an efficient way to build constant tables within a program. DATA statements provide values, the READ statement assigns the values to variables, and 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. Here are some examples:

```
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 gets "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.

Examples:

```
     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 which has the following syntax:

PRINT [#<expr>] 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" which contains specific formatting directives for each item in the "output list". The items in the output list can be constants, variables, or expressions of any atomic type. BLANKS ARE NOT ALLOWED IN FORMAT STRINGS! 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 "line up" on the left side, right side, or center of the output field. Fields are commonly right-justified in reports because it arranges them into neat 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

This 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   gives   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)

Examples:

```
PRINT USING "R8.2<",5678.123          5678.12

PRINT USING "R8.2>",12.3                12.30

PRINT USING "R8.2>",-555.9             -555.90

PRINT USING "R10.2^",-6722.4599         6722.46-

PRINT USING "R5.1","9999999"          *****
```

EXPONENTIAL FORMAT

Syntax: Ew.fj

This 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 "<" and ">" justification modes are allowed and work the same way.

Examples:

```
PRINT USING "E12.3",1234.567                 1.234E+03

PRINT USING "E12.6>",-0.001234              -1.234E-03
```

INTEGER FORMAT

Syntax: Iwj

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)
>    right justify with leading spaces and sign
^    right justify with leading sign and zeros
```

Examples:

```
PRINT USING "I4<",10                  10

PRINT USING "I4>",10                   10

PRINT USING "I4^",10                   010
```

HEXADECIMAL FORMAT

Syntax: Hwj

This format can be used to display the internal binary representation of ANY data type, using hexadecimal characters. The "w" (field width) specification determines the number of hexadecimal characters to be output. Justification modes are:

```
< left justify with trailing spaces
> right justify, leading spaces
^ center justify
```

Because the number of bytes of memory used to represent data varies according to type, the following specification make the most sense for each data type:

```
H2     boolean, byte (one byte)
H4     integer (two bytes)
H10    real (five bytes)
Hn*2   string of length n
```

Examples:

```
PRINT USING "H4",100                  00C4

PRINT USING "H10",1.5                 01C0000000

PRINT USING "H8","ABC"                414243
```

STRING FORMAT

Syntax: Swj

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 format specifications are:

```
<     Left justify (default if mode omitted)
>     right justify
^     Center justify
```

Examples:

```
PRINT USING "S8<","HELLO"      HELLO

PRINT USING "S8>","HELLO"         HELLO

PRINT USING "S8^","HELLO"       HELLO
```

BOOLEAN FORMAT

Syntax: Bwj

This format is used to display boolean data. The result of the boolean expression is converted to the strings "TRUE" and "FALSE". The specification is otherwise identical to the STRING format.

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' Include 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:

```
PRINT USING "'addr',X2,H4,X2,'data',X2,H2",1000,100   prints

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. Here are some 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"

GENERAL EXECUTION PERFORMANCE OF BASIC09

The BASIC09 multipass 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 of 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 6809'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

Because 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.

BASIC09's **REAL** (floating-point) math package provides excellent performance. A special 40-bit binary floating point representation designed for speed and accuracy, was developed especially for BASIC09 after exhaustive research. The new CORDIC technique is used to derive all transcendental functions (SIN, **TAN**, **LOG**, **EXP**, etc.). The integer shift-and-add technique is faster and more consistantly accurate than the commonly used series-expansion approximations.

Nonetheless, INTEGER operations are faster because they generally have corresponding 6809 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*2.0" the value "2.0" 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!

ARITHMETIC FUNCTIONS RANKED BY SPEED

Operation	Typical Speed (MPU Cycles)
INTEGER ADD OR SUBTRACT	150
INTEGER MULTIPLY	240
REAL ADD	440
REAL SUBTRACT	540
INTEGER DIVIDE	960
REAL MULTIPLY	990
REAL DIVIDE	3870
REAL SQUARE ROOT	7360
REAL LOGARITHM OR EXPONENTIAL	20400
REAL SINE OR COSINE	32500
REAL POWER (^)	39200

This table can be used to deduce some interesting points. For example, "value*2" is not optimum - "value+value" can produce the same result in less time because multiplication takes longer than addition. Similarly, "value*value" or "SQ(value)" is MUCH faster than the equivalent "value^2". Another interesting case is "x/2.0". The REAL divide will cost 3870 cycles, but REAL multiplication takes only 990 cycles. The mathematical equivalent to division by a constant is multiplication by the inverse of the constant. Therefore, using "x*0.5" instead is almost four times faster!

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 INTEGERs 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 SUBEXPRESSIONS

Consider the expression:

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

It is exactly the same as the expression:

```
x = x+5
```

The subexpression "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 subexpressions 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.

BASIC09 also 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 6809 which is the most powerful 8-bit processor in existence at the time of this writing. But a computer can only execute what it is told to execute, and 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.

```
PROCEDURE fibonacci
  REM computes the first ten 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=15 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:="CCXXII "
  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 that 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 pialook
  REM display PIA at address (T.F. Ritter)
  REM made understandable by K. Kaplan

  DIM address:INTEGER

  INPUT "Enter PIA address"; address
  RUN side(address)
  RUN side(ad+2)
  END


PROCEDURE side
  REM display side of PIA at address
  PARAM address:INTEGER
  DIM data:INTEGER

  (* loop until control register input strobe
  (* flag (bit 7) is set
  REPEAT \ UNTIL LAND(PEEK(address+1),$80) <> 0
  (* now read the data register
  data := PEEK(address)
  (* display data in binary
  RUN prinbyte(data)
  END


PROCEDURE prinbyte
  REM print byte as binary
  PARAM n:INTEGER
  DIM i:INTEGER

  n:=n*256

  FOR i=7 TO 0 STEP -1
    IF n<0 THEN PRINT "1";
    ELSE PRINT "0";
    ENDIF
    n:=n+n
  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)

(QUICKSORT - continued)

```
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 twenty decimal digit number, with four fractional places.

```
PROCEDURE mpadd
  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)>10000 THEN
      c(i):=c(i)-10000
      carry:=1
    ELSE carry:=0
    ENDIF
  NEXT i


PROCEDURE mpsub
  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)+10000
      borrow:=1
    ELSE borrow:=0
    ENDIF
  NEXT i


PROCEDURE mprint
  PARAM a(5):INTEGER
  DIM i:INTEGER; s:STRING

  FOR i=1 TO 5
    IF i=5 THEN PRINT ".";
    ENDIF
    s:=STR$(a(i))
    PRINT MID$("0000"+s,LEN(s)+1,4);
  NEXT i
```

(continued on next page)

(multi-precision arithmetic, continued)

```
PROCEDURE mpinput
  PARAM a(5):INTEGER
  DIM n,i:INTEGER

  INPUT "input multi-precision number: ",b$
  n:=SUBSTR(".",b$)

  IF n<>0 THEN
    a(5):=VAL(MID$(b$+"0000",n+1,4))
    b$:=LEFT$(b$,n-1)
  ELSE a(5):=0
  ENDIF

  b$:="00000000000000000000"+b$
  n:=1+LEN(b$)
  FOR i=4 TO 1 STEP -1
    n:=n-4
    a(i):=VAL(MID$(b$,n,4))
  NEXT i


PROCEDURE mptoreal
  PARAM a(5):INTEGER; b:REAL
  DIM i:INTEGER

  b:=a(1)
  FOR i=2 TO 4
    b:=b*10000
    b:=b+a(i)
  NEXT i
  b:=b+a(5)*.0001
```

```
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(8):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 8
    PRINT USING "X1,H4",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="#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 "+SrcDir+" >procmaker...dir"
       OPEN #DirPath,"procmaker...dir":READ
       CREATE #ProcPath,ProcName:WRITE
       PRINT "Function ("; Function; ")";
       INPUT response
       IF response<>"" THEN
         Function=TRIM$(response)
       ENDIF
       INPUT "Redirect Input? ",response
       IF response="y" OR response="Y" THEN
         ReInput="<" \ ELSE  \ReInput=""
       ENDIF
       INPUT "Redirect Output? ",response
       IF response="y" OR response="Y" THEN
         ReOutput=">" \ ELSE  \ReOutput=""
       ENDIF
       PRINT "Options ("; Options; ")";
       INPUT response
       IF response<>"" THEN
         Options=TRIM$(response)
       ENDIF
```

MAKEPROC - CONTINUED

```
  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
          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<>"Y" AND response<>"y"
```

```
     IF response<>"B" AND response<>"b" THEN
       BYE
     ENDIF
***************
* INKEY - a subroutine for BASIC09 by Robert Doggett

* Called by: RUN INKEY(StrVar)
*            RUN INKEY(Path,StrVar)
* INKEY determines if a key has been typed on the given path
* (Standard Input if not specified), and if so, returns the next
* character in the String Variable.  If no key has been typed, the
* null string is returned.  If a path is specified, it must be
* either type BYTE or INTEGER.

   0021               TYPE     set   SBRTN+OBJCT
   0081               REVS     set   REENT+1

   0000 87CD005E               mod   InKeyEnd,InKeyNam,TYPE,REVS
                                     ,InKeyEnt,0
   000D 496E6B65      InKeyNam fcs   "Inkey"
 D 0000                        org   0          Parameters
 D 0000               Return   rmb   2          Return addr of caller
 D 0002               PCount   rmb   2          Num of params following
 D 0004               Param1   rmb   2          1st param addr
 D 0006               Length1  rmb   2          size
 D 0008               Param2   rmb   2          2nd param addr
 D 000A               Length2  rmb   2          size
   0012 3064          InKeyEnt leax  Param1,S
   0014 EC62                   ldd   PCount,S   Get parameter count
   0016 10830001               cmpd  #1         just one parameter?
   001A 2717                   beq   InKey20    ..Yes; default path A=0
   001C 10830002               cmpd  #2         Are there two params?
   0020 2635                   bne   ParamErr   No, abort
   0022 ECF804                 ldd   [Param1,S] Get path number
   0025 AE66                   ldx   Length1,S
   0027 301F                   leax  -1,X       byte variable?
   0029 2706                   beq   InKey10    ..Yes; (A)=Path number
   002B 301F                   leax  -1,X       Integer?
   002D 2628                   bne   ParamErr   ..No; abort
   002F 1F98                   tfr   B,A
   0031 3068          InKey10  leax  Param2,S
   0033 EE02          InKey20  ldu   2,X        length of string
   0035 AE84                   ldx   0,X        addr of string
   0037 C6FF                   ldb   #$FF
   0039 E784                   stb   0,X        Initialize to null str
   003B 11830002               cmpu  #2         at least two-byte str?
   003F 2502                   blo   InKey30    ..No
   0041 E701                   stb   1,X        put str terminator
   0043 C601          InKey30  ldb   #SS.Ready
   0045 103F8D                 OS9   I$GetStt   is there an data ready?
   0048 2508                   bcs   InKey90    ..No; exit
```

```
004A 108E0001            ldy    #1
004E 103F89              OS9    I$Read      Read one byte
0051 39                  rts                return error status
0052 C1F6       InKey90  cmpb   #E$NotRdy
0054 2603                bne    InKeyErr
0056 39                  rts                (carry clear)
0057 C638       ParamErr ldb    #E$Param    Parameter Error
0059 43         InKeyErr coma
005A 39                  rts
005B 1A6916              emod
005E            InKeyEnd equ    *
```

SYSTEM MODE COMMANDS

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

EDIT MODE COMMANDS

+	<cr>	c*	l*	r*
+*	<line #>	d	q	s
-	<space>	d*	r	s*
-*	c	l		

DEBUG MODE COMMANDS

$	DEG	LET	Q	STEP
BREAK	DIR	LIST	RAD	TROFF
CONT	END	PRINT	STATE	TRON

PROGRAM 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

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				

TRANSCEDENTAL 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			

OPERATOR PRECEDENCE

highest ->	NOT	-(neg)				
	^	**				
	*	/				
	+	-				
	>	<	<>	=	>=	<=
	AND					
lowest ->	OR	XOR				

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.

Runb is the BASIC09 run-time package. It is similar to BASIC09 with the following exceptions: Runb is about half the size of BASIC09 and no file 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-Q** can be trapped by **ON ERROR GOTO** where BASIC09 can't.

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.

Packed modules can be executed without Runb, but BASIC09 will have to be used and more space will be required.

Index