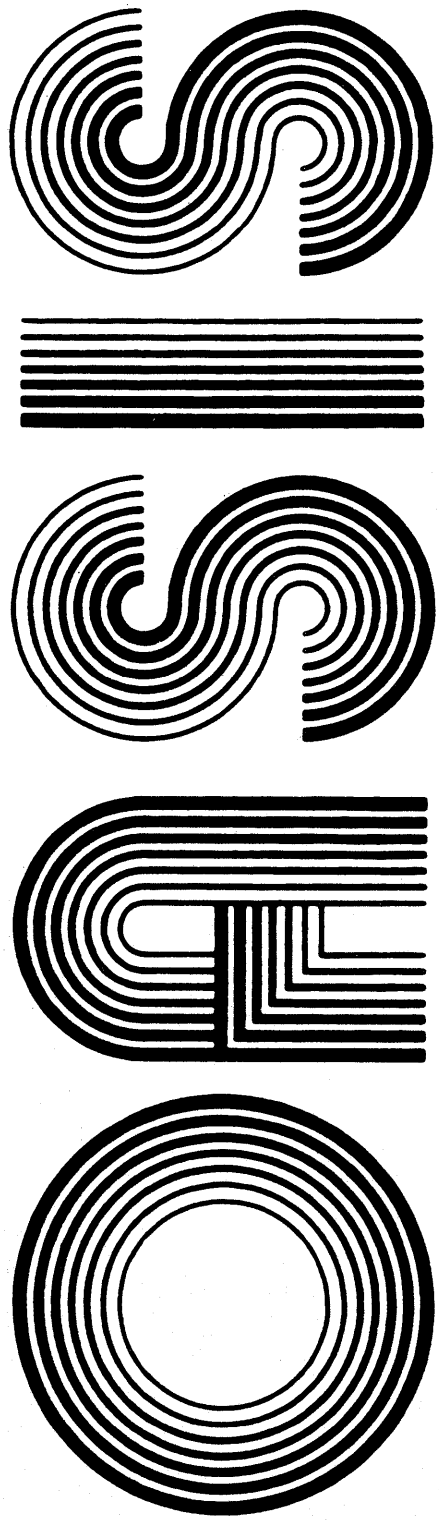
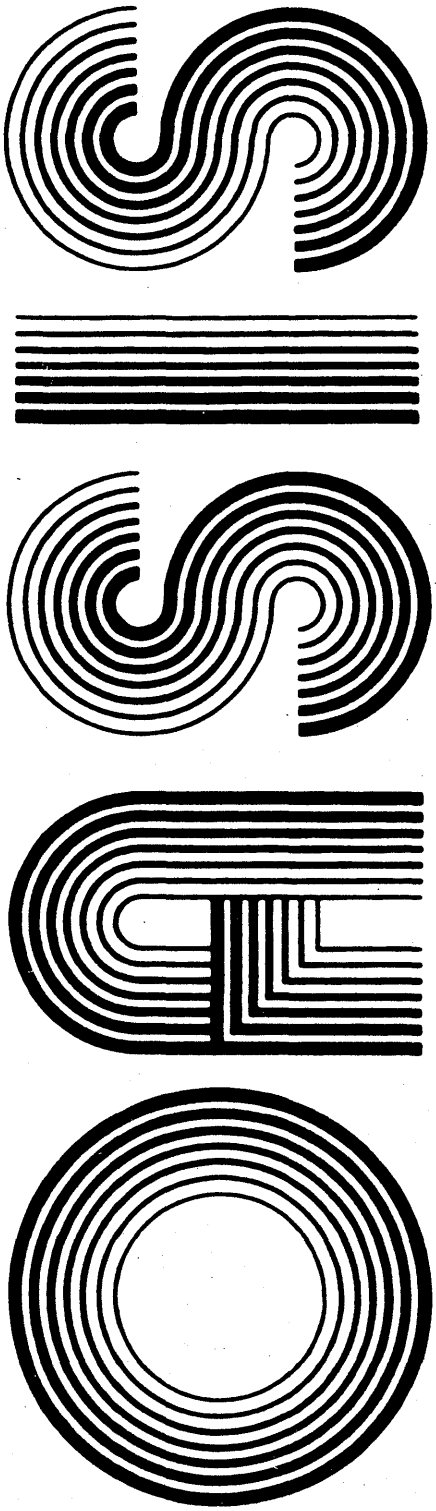


BASIC LANGUAGE REFERENCE MANUAL

OPERATING SYSTEM SOFTWARE

MAKES MICROS RUN LIKE MINIS

PHASE ONE
SYSTEMS, INC.



OPERATING SYSTEM SOFTWARE

MAKES MICROS RUN LIKE MINIS

BASIC LANGUAGE REFERENCE MANUAL

Second Edition

Revised

Documentation by: C. P. Williams
Software by: Timothy S. Williams

PHASE ONE
SYSTEMS, INC.

7700 EDGEWATER DRIVE SUITE 830
OAKLAND, CALIFORNIA 94621 USA

P R E F A C E

This manual describes the OASIS BASIC programming language interpreter/compiler available with the OASIS Operating System.

It is intended to be a reference manual, that is, the user is assumed to have general programming skills. When this is the case this manual can instruct the user on the features and uses of OASIS BASIC.

The OASIS BASIC language conforms to, and is an extension of, the American National Standard for Minimal BASIC, BSR X3.60.

The experienced BASIC programmer may find the appendices sufficient for his use. However, OASIS BASIC offers many features not found in standard Dartmouth BASIC, ANSI minimal BASIC or other dialects of BASIC.

This manual, named BASIC , like all OASIS documentation manuals, has the manual name and revision number (if applicable) in the lower, inside corner of each page of the body of the manual. In most chapters of the manual the last primary subject being discussed on a page will be identified in the lower outside corner of the page.

Related Documentation

The following publications provide additional information that may be required in the use of the OASIS BASIC language:

OASIS System Reference Manual

OASIS Text Editor Reference Manual

OASIS EXEC Language Reference Manual

OASIS MACRO Assembler Language Reference Manual

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This reference manual describes the BASIC language as implemented in the OASIS Operating System. It is an interpreter/compiler language. This means that the advantages of an interpreter exist (ability to make changes to the source program and immediately re-execute, immediate mode, etc.) along with the advantages of a compiler (faster execution, smaller program size on disk and in memory, and source program protection).

1.1 Organization of This Manual

This manual discusses each command or statement in a separate section of the appropriate chapter ("COMMANDS", "STATEMENTS", or "FUNCTIONS").

Each command or statement is described in four subsections:

1. **General form:** defines the specific syntax of the statement or command. This section is enclosed in a box at the top of the page. Also included here is a "See also" reference listing commands or statements that have a similar or related function and might be used instead of the command or statement specified.

2. **Purpose:** one or two sentences that summarize the purpose or general function of the statement or command.

3. **Comment:** detailed description of the statement or command specifying any restrictions, exceptions or errors that may occur.

4. **Examples:** general examples of the various forms of the statement or command, if applicable. Invalid examples are also included, if meaningful.

In addition, the appendices at the end of this manual give summaries of the statements, commands, functions, error messages and some general examples of BASIC programs.

1.2 Documentation Standards

In this documentation, the following standards will be used:

- All keywords are spelled out even though BASIC normally only requires the first three characters of a keyword.
- Fields enclosed with angle brackets <> are required for correct BASIC syntax.
- Fields enclosed with brackets [] are optional and not required for valid syntax.
- Fields grouped in vertical columns or separated by vertical bars indicate that all are valid forms.
- Any parenthesis shown are required for valid syntax.
- The term <CR> indicates the entry of the key CARRIAGE RETURN.

1.3 BASIC Command Modules

The OASIS BASIC and RUN programs are held on disk in several separate files. This is required due to the fact that program overlays are used. The five files containing the interpretive BASIC program must all reside on one disk. The four files containing the RUN program must both reside on one disk. The files and their primary functions are as follows:

BASIC.COMMAND	Initialization and set up.
BASIC.OVERLAY1	Editor and syntax analyzer.
BASIC.OVERLAY2	Compiler.
BASIC.OVERLAY3	Cross-reference generator.
BASIC.OVERLAY4	Program execution and debugger.
BASIC.LOADFILE	Re-entrant run time module (multi user only).
BASIC2.LOADFILE	Re-entrant run time module for RUN2 (multi user only).
RUN.COMMAND	Initialization and set up.
RUN2.COMMAND	Initialization and set up for RUN2.
RUN.OVERLAY1	Run time monitor (single user only).
RUN.OVERLAY2	Run time monitor for RUN2 (single user only).

1.4 BASIC Program File Types

OASIS BASIC uses three different file types for programs. These file types are BASIC, BASICOBJ, and BASICCOM.

A BASIC program with a file type of BASIC is a file in ASCII format and is usable by TEXTEDIT and EDIT as text files. A program with a type of BASIC may be loaded with the BASIC interpreter but may not be RUN, CHAINed, or LINKed to.

A BASIC program with a file type of BASICOBJ is a file generated by the BASIC interpreter after the program has been syntax checked. This type of a file cannot be used by other system programs (except COPYFILE). A BASICOBJ file is a program that is "pseudo-compiled", that is, all keywords have been coded to reduce the storage requirements and to increase execution speed. Even though this type of a file is pseudo-compiled it is still listable by BASIC and still has all remarks, variable names and line labels in it. This file type should be used for all source programs and is the default type used by the SAVE command.

A BASIC program with a file type of BASICCOM is a file generated by the BASIC compiler after the program has been compiled. This type of file cannot be used by other system programs (except COPYFILE) and may only be executed with the RUN command. A BASICCOM file is a program that has had all remarks removed from it and all variable names have been reduced to codes (variables defined as COMMON are not affected) and all line label references have been changed to a shorter and faster method of branching.

1.5 Loading BASIC

The BASIC command is a program module that is accessed by the Operating System through the OASIS Command String Interpreter (CSI). After CSI has displayed it's prompt character (>), the operator may enter a BASIC command as described below. CSI will load the BASIC interpreter and enter the edit mode, allowing you to load, execute, or edit any BASIC program.

In the edit mode of BASIC, acceptable input is a command, an immediate statement, or a numbered statement. An immediate statement is one that is executed immediately, a numbered statement is stored in memory for execution at a later time.

1.6 BASIC and RUN Commands

The OASIS BASIC command allows the user to access the BASIC interpreter/compiler to create, change, debug, execute or compile BASIC programs.

1.6.1 BASIC Interpreter

The OASIS BASIC interpreter is invoked by using the command BASIC. The format of the interpretive BASIC command is:

BASIC [(COMMA[)]]

Where:

COMMA Indicates that numbers input or output during execution of BASIC programs are to use the European format for number representation (see below).

When the BASIC interpreter is invoked the prompt character for BASIC will be displayed (-) and you will be allowed to enter any valid BASIC command to develop, test, or execute your interpretive programs (file type of BASICOBJ or BASIC).

1.6.2 BASIC Compiler

The OASIS BASIC compiler is invoked by using the same command BASIC as the interpreter but with the additional specification of the program name to be compiled. The format of the BASIC compiler command is:

BASIC [<program-name> [(<compile options>[)]]

Where:

program-name Specifies the name of a BASIC program file that the user wishes to compile. This operand has the format <fn[.ft][:fd]>, where:

fn Indicates the file name of the BASIC program to be compiled. If omitted (i.e., the entire program-name operand is omitted), then

BASIC is invoked in the immediate mode.

ft Indicates the file type of the program to be compiled. Only BASIC programs with file type BASICOBJ can be compiled.

Note: Programs written and compiled under a version of BASIC prior to version 5.5 may be recompiled by specifying the file type of BASICCOM. This feature allows those users who do not have the source (BASICOBJ) to programs to compile them with the current version of BASIC.

fd Indicates the label of the directory or the name of the disk that the program file resides on. When omitted the normal search sequence for user programs is used.

BASIC Compile Options

PRINTER[n] Indicates that the program listing is to be output to the printer.

LIST Indicates that the program listing is to be output to the console.

NOLIST Indicates that no program listing is to be output to the console. This is a default option.

XREF Indicates that the cross reference table is to be generated and output with the program listing.

NOXREF Indicates that no cross reference table is to be included in the program listing. This is a default option.

OBJ=drv Indicates that the compiled program is to be output to the specified drive. When this option is not specified and the NOOBJ option is not specified the compiled program will be output to the same drive as the source program currently resides on.

NOOBJ Indicates that the compilation is for test or listing purposes only--no compiled program will be output to disk.

NOTYPE Indicates that the compilation is to be performed in 'silent' mode--display of compile statistics is suppressed--only errors are displayed if encountered.

Compiled programs are saved on disk with file type BASICCOM.

The COMMON statement must be used to specify variables that are used by more than one segment of a compiled program.

The program listing generated by the BASIC compiler (option PRINT or LIST) includes the relative address in hexadecimal of each statement listed. This information is important in case an error is encountered during execution of the compiled program. Since line numbers are removed by the compiler the location of an error is indicated by its relative address.

At the end of the compilation process statistics about the program compiled are displayed on the console (unless option NOTYPE is specified). This display included the following information:

```
Oasis BASIC compiler ver n.n (date) statistics.
Input source lines:      nnnn
Input source size:       nnnnn
Output object size:      nnnnn
Source reduction:        nn%
Compiler errors:         nnn
String variables:        nnnn
Float variables:         nnnn
Integer variables:       nnnn
Compile rate:            nnnn lines per minute.
```

1.6.3 Execution of Compiled Programs

The OASIS RUN command allows the user to execute a compiled, BASIC program. The format of the RUN command is:

RUN <program name> [(<option>[])]

Where:

program-name Specifies the file name of the program to be executed. The file type may be specified, however it must be BASICOBJ. The file disk may be specified but when it is not specified all attached disk drives will be searched for the program.

RUN Options

COMMA Indicates that numbers input or output during execution of the compiled programs are to use the European format for number representation (see below).

TRACE Indicates that the hexadecimal address of each statement executed is to be displayed on the console. These hexadecimal addresses are the same as those listed when the program was last compiled with a TYPE or PRINT option.

The RUN command can only execute compiled programs--there is no immediate or command mode available to the user when this command is in control.

A smaller version of BASIC may be used to execute a compiled program. This smaller version is invoked by used the command RUN2 instead of RUN. Refer to the appendix on "RUN2 Statement and Function Exceptions" in the back of this manual.

BASIC Prompting Character

When the BASIC interpreter/compiler is in immediate or command mode a hyphen (-) is displayed at the left side of the terminal, indicating that BASIC is awaiting a command. This is the prompting character for OASIS BASIC.

1.7 European Format for Numbers

The OASIS BASIC interpreter and run time monitor support the European format for number representation for input and output of numbers in ASCII format (i.e., to/from the console). The European format is invoked by using the COMMA option on the command line to BASIC or RUN or by using the OPTION COMMA statement from within the program being executed.

When the European format is invoked all input and output of numbers in their ASCII format (INPUT and PRINT statements) will conform to the European standard for the remainder of the session in BASIC or RUN. The only way to revert to the 'American' format is to exit from BASIC or RUN and return control to the operating system.

The European format, as used by OASIS BASIC, denotes a 'decimal point' with the character comma (,) and denotes the division of thousands with the decimal character (.). This is the exact opposite of the 'American' format. An additional convention used in OASIS BASIC is the separation of elements in a list of numbers is the semicolon character (;).

For some clarification examine the following examples:

American format	European format
1	1
1,000	1.000
10.23	10,23
1,000,000.00	1.000.000,00
1.1 , 2.4 , 3.5	1,1 ; 2,4 ; 3,5

As stated above, the COMMA option applies to all input and output of ASCII numbers. This means that a file created with PRINT statements when the COMMA option was not in effect will not be input properly with the COMMA option in effect.

In addition to input and output of ASCII numbers the COMMA option affects the operation of those functions that operate on numbers in ASCII format. The functions affected include: NBR, STR$, and VAL.

The operation of the COMMA option does not change how programs are coded; i.e., the

PRINT USING mask is coded using the 'American' format.

CHAPTER 2

FEATURES OF THE LANGUAGE

2.1 Data Files

OASIS BASIC supports four types of files: Sequential, Direct, Indexed Sequential, and Keyed. All four file types may contain ASCII data or binary data, depending upon how the individual records were created. Good programming practice will restrict you from mixing ASCII and binary data within one file. Records created with the PRINT statement are ASCII and may only be read with the INPUT or LINPUT statements. Records created with the WRITE statement are binary and may only be read with the READ or READNEXT statements.

SEQUENTIAL files are variable in length with variable length records--only as much space is used as is needed. Sequential files are accessed sequential, from the beginning of the file to the end. In order to read any specific record in the file all preceding records must be read.

DIRECT files are fixed in size with fixed length records. This file type is accessed randomly by relative record number.

INDEXED Sequential files are similar to DIRECT files in that they are fixed in size and length and the records are accessed randomly but with an ASCII key or index. This file type also allows you to read the file sequentially, in the ASCII collating sequence of the keys.

KEYED files are similar to INDEXED with the exception that the file, when accessed sequentially, is in the physical sequence of the file on disk, not in ASCII collating sequence.

2.2 Cursor Control

Many types of terminals are "known" to BASIC and their various types of cursor control are handled by common functions (AT and CRT). Refer to the OASIS System Reference Manual, "ATTACH COMMAND", and the "Terminal Class Codes" Appendix for details.

2.3 Chaining and Linking

Chaining and linking allows very large programs to be segmented for execution in a system with a relatively small amount of memory. Chaining transfers control to the named segment and closes all open files. Linking transfers control to the named segment without closing any files.

2.4 User Defined Control Keys

User programs can test whether one of several different control keys were entered as input, and take appropriate action in the program. Refer to the chapter on User Definable Keys in this manual for details.

2.5 Compatibility

BASIC is upward compatible with Dartmouth BASIC and conforms to the American National Standard for Minimal BASIC, BSR X3.60.

2.6 Other Features

OASIS BASIC provides many other features not normally found in other micro-computer BASICs such as:

- Multiple statements on one line.
- Multiple line user defined functions (DEF FN-FNEND).
- Line length of up to 255 characters.
- Long variable names.
- Line labels.
- Error trapping (ON ERROR GOTO).
- Complex IF THEN ELSE statements.
- Multiple line IF-IFEND structure.
- Multiple line WHILE-WEND structure.
- Multiple line SELECT-CASE-OTHERWISE-CEND structure.
- String handling with string length of up to 255 characters.
- String arrays.
- Matrix (array) input/output and assignment.
- Formatted output (PRINT USING).
- Formatting function.
- Formatted input (LINPUT USING).
- European format for numbers.
- Interface to user written assembly subroutines (USR).
- Interface to system commands (CSI).
- Interface to any device (GET, PUT, WAIT).
- Bit manipulating logical functions.
- Thirteen digit precision BCD (binary coded decimal) arithmetic.
- Floating point values in range 10^126 to 10^-126
- Integer arithmetic (-32767 to +32767).
- Program debugging aids such as single step, break-points, etc.
- Automatic line number entry.
- Syntax analysis on statement entry.
- Extensive set of string functions.
- Compile option to compress and protect the program.
- Cross reference listing of variables.

CHAPTER 3

BECOMING FAMILIAR WITH BASIC

This section assumes that you have loaded BASIC and that you have received the BASIC prompting character (-), indicating that BASIC is waiting to perform whatever instuction you give. In order to make the most efficient use of your sessions with BASIC, you need to know several things about communicating with the system.

For the time being the specific statement, command and line syntax will be ignored. These concepts are discussed in the next chapter.

You will communicate with the system by using its primary input/output (I/O) device, called the CONSOLE TERMINAL. This device will include either a printing mechanism or a video screen (CRT), as well as a keyboard, similar to that found on a typical electric typewriter. On a console terminal keyboard, however, there are a few symbols and extra keys which may be new to you. Note the position of "extra" keys, especially the ones marked "CONTROL" (or "CTRL", or "CNTRL" or something similar), "RETURN" (or "CARRIAGE RETURN", or "NEW LINE" or something similar), "RUBOUT" (or "DEL", or "BACKSPACE", or something similar), "ESC" (or "ESCAPE", or "ALTMODE").

3.1 Some Basic BASIC Concepts

OASIS BASIC has two modes of operation:

IMMEDIATE MODE or command mode, in which lines typed to the system are executed without delay;

EXECUTION MODE or program mode, in which the system executes instructions which have been stored previously in the form of a PROGRAM.

Prior to learning how to work with BASIC in these modes, you must understand certain concepts and terminology, which are explained in this section.

A COMMAND is a special type of BASIC instruction which may be executed in immediate mode, not as part of a program. Commands generally provide services which are not meaningful or useful while a program is executing.

For example, the command LIST generates a listing of the program currently in the BASIC program/data area of memory. (This is called the CURRENT PROGRAM.) It is a rare application which requires a program to list itself, and so the LIST function is a command.

A STATEMENT is a BASIC instruction which may be used as part of a PROGRAM or in IMMEDIATE MODE. Typical among statements is PRINT, which causes information to be output to the console terminal. Statements begin with a VERB from which the statement derives its name. The verb may be followed by ARGUMENTs and keywords. An argument is a piece of information on which the statement operates, or which is used to modify the operation of the statement. For example, the string literal "HI" is the argument of the following statement:

```
PRINT "HI"
```

A BASIC program is structured as a sequence of LINEs, each containing one or more statements. A line starts with a LINE NUMBER, which is an INTEGER (that is, a whole number) in the range of 1 to 9999. A statement follows the line number, and the combination is called a PROGRAM LINE. A typical line is:

```
70 PRINT "THIS IS ONE STATEMENT."<CR>
```

More than one statement may exist on a program line, as long as individual statements on that line are separated by a backslant (\) character. Here is an example of a multiple-statement program line with three statements:

```
100 LET A = 0 \ LET B = 1 \ PRINT A,B
```

All statements may be executed in immediate mode in order to get immediate results. This is accomplished by typing a statement without preceding it with a line number. Such a statement is called an IMMEDIATE STATEMENT, and is executed as soon as it has been completely typed (indicated by striking the RETURN key). For example, if you type:

```
PRINT 3+3<CR>
```

into BASIC, you will immediately get back 6 on the terminal. This ability to execute statements in immediate mode greatly facilitates debugging by allowing you to examine (PRINT) and modify (LET) the contents of variables when a bug occurs.

Each command and statement has its own rules as to what constitutes its proper syntax and when it can be used correctly.

3.2 BASIC Uses Upper Case

BASIC requires that the instructions it executes be in upper case characters. To facilitate this, instructions typed in BASIC are translated to upper case before being stored for execution. For example, the following line is typed to BASIC:

```
10 if a1 > 25 then print "Greater than" else goto 100<CR>
```

That line is stored in memory in the following format:

```
10 IF A1>25 THEN PRINT "Greater than" ELSE GOTO 100
```

Note that all of the "keywords" have been translated to uppercase but the literal is left as is. Because of this you will not have to worry about the case mode of the instructions you type.

3.3 Typing to BASIC

Try typing some nonsense to BASIC:

```
-ABCDEFGHIJK<CR>
```

Be sure to strike the RETURN key after you finish typing a line to BASIC, as denoted by the <CR> symbol above. This is the signal for BASIC to accept and process what you've typed. If you fail to strike the RETURN key, BASIC will patiently wait forever for you to type more!

BASIC should respond to your nonsense with the message:

```
Unrecognized command
```

In general, this message is BASIC's way of saying "I don't understand you". It usually means that you typed the right thing incorrectly, or (as in this case) the wrong thing altogether. This is an example of an ERROR MESSAGE. Such messages are sent to you in order to alert you to any difficulties which BASIC encounters as it attempts to carry out your instructions. The error message should provide a clue as to the nature of the problem, and imply the possible steps you might use to correct it. (Correcting computer problems is called "debugging". A problem itself is referred to as a "bug".)

Let's type something which BASIC will understand:

```
-PRINT 25/2<CR>
```

(Remember that the <CR> means to strike the RETURN key.)

You should get the answer displayed on the terminal.

Commands may be entered with abbreviations (such as LEN for the LENGTH command) but incorrect syntax or spelling errors will not be allowed and you will have to re-enter the command.

Statements (immediate or stored) may also use abbreviations for the statement verb (such a PRI for PRINT). Statements, different from commands, do not have to be re-entered to correct spelling or syntax errors, just modified to the correct form.

For example, try typing the statement:

```
10 FOR I=1TOX<CR>
```

BASIC will respond by displaying:

```
Keyword Missing or mis-spelled
0010 FOR I=1TOX
```

The underscore character will be used to identify the cursor position. BASIC is "saying" that it recognizes the statement as a FOR statement but it can't find the keyword TOX. This is due to the fact that variable names may be more than one character long and the letters TOX could be a variable name. You must surround keywords and verbs with some delimiting character, usually a space.

To correct the error in this statement, enter an I, space, <CR>, space, space, I, space, <CR>, <CR>. This is explained below.

When an error is detected by the syntax analyzer the error message is displayed as above and an implied MODIFY command is executed with the cursor pointing to the location of the error. The correction just specified causes MODIFY to go into insert mode (the I character), insert a space at that location, exit the insert mode (the <CR>), advance two places (the space, space characters), go into insert mode again and insert a space, exit the insert mode, then exit the modify mode (the last <CR>).

The following display illustrates this correction:

```
0010 FOR I=1TOX                          Enter I
0010 FOR I=1 TOX                         Enter space
0010 FOR I=1  TOX                        Enter <CR>
0010 FOR I=1 TOX                         Enter space
0010 FOR I=1 TOX                         Enter space
0010 FOR I=1 TOX                         Enter I
0010 FOR I=1 TO X                        Enter space
0010 FOR I=1 TO X                        Enter <CR>
-                                        Enter <CR>
                                         Enter <CR> (display command)
0010 FOR I = 1 TO X
-
```

When you exit the implied MODIFY command the syntax of the statement is re-examined for errors. If no more errors are detected the statement is saved (or executed if an immediate statement) and control of BASIC returns to the mode it was in (in the above case it returns to the command mode).

As another example, consider the following:

```
-10 PRI SQR(23;NOW IS THE TIME;A$B$<CR>
Missing parentheses
0010 PRI SQR(23;NOW IS THE TIME;A$B$          Enter I
0010 PRI SQR(23;NOW IS THE TIME;A$B$          Enter )
0010 PRI SQR(23);NOW IS THE TIME;A$B$         Enter <CR>
Comma required
0010 PRI SQR(23);NOW IS THE TIME;A$B$         Assumes NOW is variable name
0010 PRI SQR(23);NOW IS THE TIME;A$B$         Enter back space
0010 PRI SQR(23);NOW IS THE TIME;A$B$         Enter back space
0010 PRI SQR(23);NOW IS THE TIME;A$B$         Enter back space
0010 PRI SQR(23);NOW IS THE TIME;A$B$         Enter I
0010 PRI SQR(23);"NOW IS THE TIME;A$B$        Enter "
0010 PRI SQR(23);"NOW IS THE TIME;A$B$        Enter F;
0010 PRI SQR(23);"NOW IS THE TIME;A$B$        Enter I
0010 PRI SQR(23);"NOW IS THE TIME;A$B$        Enter "
0010 PRI SQR(23);"NOW IS THE TIME";A$B$       Enter <CR><CR>
Comma required
0010 PRI SQR(23);"NOW IS THE TIME";A$B$       Enter I
0010 PRI SQR(23);"NOW IS THE TIME";A$B$       Enter ;
0010 PRI SQR(23);"NOW IS THE TIME";A$;B$      Enter <CR><CR>
-                                             Enter <CR>
  10 PRINT SQR(23);"NOW IS THE TIME";A$;B$
-
```

3.4 Consistency in Listing

Because OASIS BASIC is an interpreter/compiler it saves statements in a compact, coded format. When a program listing is requested (or even a single line displayed) the coded format must be expanded to a display format. It does this expansion in a very consistent manner--consistency is desirable in programming:

- All keywords and verbs are always spelled out fully.
- All keywords and verbs are surrounded by spaces.
- Multi-statement line separators are surrounded by spaces.
- Lists of variables, expressions, and line references are separated by their proper punctuation.
- I/O channel specifications are surrounded by spaces.
- Commas are added when the statement syntax requires.
- Expressions are displayed without any embedded spaces.
- The assignment operator is surrounded by spaces.
- Any leading spaces in a line are maintained.
- String literals are always surrounded with the double quotation mark character (").

For example, the following is performed (entry and display):

```
-AUTO<CR>
  10 REM This is a remark<CR>
  20       IF    (A$ > B$) * 5/ (5+ VALUE%) THEN GO SUB 1000\STOP<CR>
  30       A=23 * B<CR>
  40<CR>
-LIST<CR>

  10 REM This is a remark
  20       IF (A$>B$)*5/(5+VALUE%) THEN GOSUB 1000 \ STOP
  30       A = 23*B

-
```

CHAPTER 4

PROGRAMMING IN BASIC

4.1 Structure of a BASIC Program

A BASIC program consists of a set of statements constructed with the language elements and syntax described in the following chapters. Expressions, line numbers, labels, and statements are joined to solve a particular problem, with each line containing instructions to BASIC.

4.1.1 Syntax

Syntax is a term refering to the structure of the parts of a statement and the punctuation characters separating those parts. As an example, the syntax of a sentence in the English language is: <subject> <verb> <object> <punctuation>. Unfortunately for elementary school children (and university professors) the syntax of sentences has many acceptable variations with each variation having variations and options and exceptions.

On the other hand, computer languages are very structured with very specific syntax requirements for each statement (sentence). There may be options to the structure but there are no exceptions.

4.1.2 Character Set

OASIS BASIC uses the full ASCII (American Standard Code for Information Interchange) character set for its alphabet. This set includes:

- Letters A through Z
- Letters a through z
- Numbers 0 through 9
- Special characters (see ASCII character set in appendix).

This character set enables you to include any ASCII character as part of a program. BASIC translates the characters that you type into machine language; some characters are processed and some are left as entered.

The BASIC editor translates characters in the following manner:

- Letters A through Z - left as entered.
- Letters a through z - left as entered if in a statement remark or string literal (enclosed in quotation marks); translated to upper case equivelent in all other contexts.
- Non displayable characters (BELL, DC1, FS, etc) - ignored.
- Other control characters -
  BS treated as editing character (backspaces one position); is not entered into the actual line.
  HT when entered after line number and before the start of the statement: translated to five (5) spaces; when entered in middle of statement translated into one space.
  LF ignored.
  VT ignored.
  FF ignored.
  CR treated as end-of-line character. In auto entry mode the next line number will be displayed.
- Special characters:
  ; When entered at start of statement is translated into REM.
  Statement separator for multi-statement line.

4.1.3 Line Format

The general format of a program line is as follows:

```
line number   label      verb        operand

     1010     LABEL:     PRINT       SQR(X^2+Y^2)
```

All lines in a BASIC program must begin with a line number. This number must be a positive integer within the range of 1 through 9999. A BASIC line number is a label that distinguishes one line from another within a program and determines the placement of that line in the program.

Leading zeroes (as well as leading and trailing spaces) have no effect on the number. However, you cannot have embedded spaces within a line number.

4.2 Statements

BASIC statements consist of keywords that you use in conjunction with the elements of the language set: constants, variables, and operators. These statements divide into two major groups: executable statements and non-executable statements.

At least one space or tab must follow all statement keywords in order for BASIC to recognize the keyword as such. For example:

```
Acceptable          10 PRINT CUR.DATE$
Unacceptable        10 PRINTCUR.DATE$
```

Some keywords consist of two words such as PRINT USING, ON ERROR, MAT INPUT. These keywords must also be separated by at least one space or tab character. Two exceptions to this are the GO TO and GO SUB keywords. It is acceptable to use the keywords GOTO or GOSUB without a separating space.

Statement keywords are reserved, and therefore, cannot be used as a variable name (see appendix "Reserved Keywords"). However, keywords can be used as line labels.

4.2.1 Single Statement, Multi-Statement Lines

You have the option of typing either one statement on one line or several statements on one line.

A single statement line consists of:

- A line number (from 1 to 9999).
- An optional line label followed by a semicolon (:).
- A statement keyword.
- The body of the statement.
- A line terminator.

This is an example of a single statement line:

```
10 PRINT A,BETA*TODAY+3.
```

To enter more than one statement on a single line (multi-statement line), separate each complete statement with a backslant (\). The backslant symbol is the statement separator. You must type it after every statement except the last in a multi-statement line. For example, the following line contains three complete PRINT statements:

```
10 PRINT ALPHA$;BETA; \ PRINT CUR.DATE$ \ PRINT "Total =";TOTAL
```

The line number labels the first statement in a line. Consequently, you must take this into consideration if you plan to transfer control to a particular statement within a program. For instance, in the previous example, you cannot execute just the statement

```
PRINT CUR.DATE$
```

without executing PRINT ALPHA$;BETA; and PRINT "Total =";TOTAL

All executable statements can appear in a multi-statement line.

The rules for structuring a multi-statement line are:

- Only the first statement in a series has a line number.
- Only the first statement in a series can have a line label.
- Successive statements must be separated with a backslant.

4.3 Line Labels

All OASIS BASIC lines have line numbers and the line may be referenced by other statements using the line number of the line (GOSUB, GOTO, etc.). Lines may also have a line label.

Line labels are useful for referencing lines when the line number is unknown, when you wish to "document" the function of a line or sequence of lines, etc.

A line label consists of one or more letters, digits, or periods with the first character being a letter. There is no limit on the length of a line label but you should use labels that are short, but still meaningful (you have to type the entire label each time it is referenced).

A line label must be unique within a program. When a line label is defined it must precede any statements on the line and be separated from the first statement by the colon character (:).

The following lines are all acceptable uses of line labels:

```
10 MAINLINE: WHILE CONTROL = 0
20 GOSUB INPUT.ROUTINE
30 IF INPUT$ = "" GOTO ERRORS
40 INPUT.ROUTINE: REM Subroutine to accept input
```

It is permissible for a line label to be a keyword (no confusion arises due to the context in which a line label appears), however a label may not start with the letters REM.

4.4 Documenting Procedures

BASIC allows you to document your methods, insert notes and comments, or leave yourself messages in the source program. This type of documentation is known as a remark or comment. There is only one way of inserting comments within a BASIC source program: the REM statement.

BASIC ignores anything in a line following the keyword REM including a backslant character. The only character that ends a REM statement is a line terminator. Therefore, a REM statement must be the only statement on a line or the last statement in a multi-statement line.

```
10 LET A=B  REM Variable A receives current value of B
```

You can use the semicolon character (;) instead of the keyword REM. BASIC will translate this into the keyword REM and display it as such whenever a listing is produced.

You can use the line number of a REM statement in a reference from another statement, i.e. GOSUB.

Another method of documentation, used in conjunction with remarks, is indentation. Any spaces or tabs entered between the line number and the first character of the line will be maintained by BASIC for listing purposes. This allows you to show the structure or hierarchy of the program.

Remarks and/or leading spaces have no impact on a program after it is compiled (one of the functions of compilation is to remove these from the program).

Refer to the appendix containing program examples for illustrations of documentation techniques.

4.5 Entering and Modifying Programs

OASIS BASIC allows programs to be entered, debugged, and modified while in the BASIC environment. Refer to the chapter on "BASIC Commands" for information on the use of the commands in editing a program (AUTO, CHANGE, DELETE, DISPLAY, DOWN, LIST, LOCATE, MODIFY, and UP).

It is important to note that BASIC performs syntax analysis when the statement is entered, not when the statement is executed. This not only increases the speed of execution but also prevents any syntax errors from being entered. The main advantage of this pre-execution syntax analysis is that the program is free of all syntax errors even though some of the lines in the program have never been executed.

CHAPTER 5

ELEMENTS OF THE BASIC LANGUAGE

In order to write programs in BASIC you must be familiar with the terms and phrases used to describe the program elements. You will probably recognize most of these terms from previous experience; however, the following sections define these terms within the context of OASIS BASIC.

5.1 Constants

A constant is an element whose value does not and cannot be changed during the execution of a program.

There are three types of constants in BASIC:

- Numeric (also called floating point numbers)
- Integer (whole numbers)
- String (alphanumeric and/or special characters)

5.1.1 Numeric Constants

A numeric constant is one or more decimal digits, either positive or negative, with a decimal point specified. (The decimal point may be omitted when the constant is a whole number outside of the range +32767 to -32767.)

The following are all valid numeric constants:

25.	3.14159	234567
-1234.01	.000002	32760.
12345678901.23	-9876543210123	.1234567890123

Numeric constants cannot contain any embedded space characters.

BASIC accepts and maintains numeric constants within a range of 13 significant digits.

When you type a numeric constant with more than 13 significant digits specified the excess, least significant digits will be truncated.

It is possible to enter and maintain a number that is outside the range of precision by using an alternate format:

<+ or ->x.xxxxxxxxxxxxE<+ or ->nnn

Where:

<+ or -> is the sign of the number. The plus sign is optional with positive numbers; the minus sign is required with negative numbers.

x is the number with up to 13 significant digits.

E represents the words "times 10 to the power of"

nnn is the exponential value (the power of 10) in the range of +126 to -126

This method of mathematical shorthand is called E format, floating point notation, or scientific notation. It is BASIC's way of representing scientific notation. To use this format, append the letter E to the number, follow the E with an optionally signed integer constant. This constant is the exponent--it can be 0 but never blank.

The following are all valid numeric constants, E format:

1.2568E10	8.254681325257E-120	1235E-30
-1.234567890123E-126	2358.256824798E2	1.2E60

All E notation numeric constants are normalized after entry, that is, the decimal point (and the nnn value) is adjusted to be after the first significant digit. For example, entry of the constant 12345.58E10 will be normalized to be 1.234558E+014. If a number entered in E notation can be expressed in normal notation, it will be. For example, entry of the constant 1.25E6 will be printed as 1250000.

5.1.2 Integer Constants

An integer constant is a special type of numeric constant that is a whole number (no fractional part) written without a decimal point and in the range of +32767 to -32767. For example, the following numbers are all integer constants:

1	0	-1234
25	-15	100
32767	-32767	10000

Integers, though normally entered in decimal format (base 10) may be entered in hexadecimal format (base 16). When this is done the integer constant must be terminated with the letter H. Hexadecimal values may use the digits 0 through 9 and the letters A through F. A hexadecimal constant must start with a digit (use a zero if necessary).

The following are all acceptable hexadecimal integer constants:

1234H	0ABH	245H
0FFFFH	-1234H	0FH

The following are all unacceptable integer constants:

12AB	Invalid decimal or missing "H"
0FFFGH	G is not valid hexadecimal character
123456	Outside of range of integer
1.24	Not an integer
12E10	Outside of the range of an integer

5.1.3 String Constants

A string constant (also called a string literal) is one or more alphanumeric and/or special characters, enclosed in a pair of double quotation marks (") or single quotation marks ('). Include both the starting and ending delimiters when typing a string constant in a program. These delimiters must be of the same type (both double quotation marks or both single quotation marks).

Each character in a string constant can be a letter, a number, a space, or any ASCII character except a line terminator. The value of the string constant is determined by all of its characters. BASIC maintains every character between the delimiters exactly as you entered it into the source program.

BASIC does not normally print the delimiting quotation marks when a string constant is printed on the console, printer, or file.

Quotation marks may be included as part of the text of a string constant by either: using the opposite type of delimiting quotation marks (i.e. single within double, double within single); or by doubling the embedded quotation mark ("" or '').

The following are all acceptable string constants:

String constant	Internal representation
"This is a string constant"	This is a string constant
'This is also a string constant'	This is also a string constant
"Look at Spot's spots."	Look at Spot's spots.
'Look at Spot''s spots.'	Look at Spot's spots.
"He said, ""Open the book.""	He said, "Open the book."

5.2 Variables

Variables differ from constants in that their values may change during the execution of the program. For this reason variables are refered to by their name, not their current value. BASIC uses the most recently assigned value of a variable when performing calculations. This value remains the same until a statement is encountered that assigns a new value to that specific variable.

BASIC allows three types of variables:

- Numeric variables (name terminated with letter, digit, or period)
- Integer variables (name terminated with %)
- String variables (name terminated with $)

The type of a variable is determined by the name of the variable. BASIC allows variable names to be of unlimited length (a reasonable maximum is about two hundred characters due to the line length restriction of 255 characters).

Variable names for the three types of variables have a common syntax:

* First character must be a letter (A - Z)
* Subsequent characters are optional and may consist of letters (A - Z), digits (0 - 9) or the period character (.).
* The space character cannot be used as part of a variable name.
* The variable name cannot be a reserved word.

The following are all acceptable variable names:

TOTAL	SUM	INTEREST
SUB.TOTAL	SUM1	PRIME.INTEREST
SUB.TOTAL1	CUST.NAME$	P.INT
A	INDEX%	B0

The following are all unacceptable variable names:

123A	Must start with letter
A$ONE	Only special character allowed is period
PRINT	Reserved word
SQR	Reserved word

A variable name is identified as one of the three types of variables by a terminating type character. This type character is part of the name and makes the name different from a variable name with a different type character. For example, the following three variable names each refer to a different variable:

CUSTOMER	(numeric variable)
CUSTOMER%	(integer variable)
CUSTOMER$	(string variable)

5.2.1 Numeric Variables

A numeric variable is a named location in which a single numeric value is stored. Numeric variables contain numeric (floating point) values. A numeric variable is identified by a variable name (discussed above) without a terminating type character (last character is a letter, digit, or period).

The following are all acceptable numeric variable names:

A	A1	B9
COUNT	INDEX	RECORD.NUMBER
MAXIMUM	MINIMUM	TOTAL

The following are all unacceptable numeric variable names:

6	TOTAL-TALLY	RECORD*COUNT
9TOTAL	1A	TWO/3

When a numeric variable is first defined its value is set to zero (0). Execution of the RUN instruction clears all variables. If you require an initial value other than zero you must assign it with the LET statement.

> Note: Because other BASIC languages may not set all variables to zero before program execution you should not rely on this feature. Good programming practice dictates that you initialize all variables at the beginning of the program.

5.2.2 Integer Variables

An integer variable, similar to a numeric variable, is a named location in which a single integer value is stored. Integer variables contain integers (whole, non-fractional values). An integer variable is identified by a variable name (discussed above) with a terminating type character of a percent (%) symbol.

The following are all acceptable integer variable names:

A%	A1%	INDEX%
RECORD%	RECORD.NUMBER%	CODE%

The following are all unacceptable integer variable names:

A	B2	1TOTAL%
TOTAL1$	NAME$ONE	REC.INDEX

When an integer variable is first defined its value is set to zero (0). Execution of the RUN instruction clears all variables. If you require an initial value other than zero you can assign it with the LET statement.

An integer variable always contains an integer value (see integer constants for restrictions). If a numeric constant or variable is assigned to an integer variable, BASIC first truncates the fractional part of the floating point number. If the resulting whole number is outside the range of an integer (+32767 to -32767) the number is set to 32767 with the proper sign and an error occurs (refer to the ON ERROR GOTO statement and the appendix on error codes).

When you assign an integer variable or constant to a numeric variable BASIC will print the numeric value as an integer but maintains it as a floating point number internally.

5.2.3 String Variables

A string variable is a named location in which a single alphanumeric string of characters is stored. A string variable is identified by a variable name (discussed above) with a terminating type character of the dollar sign ($).

The following are all acceptable string variable names:

A$	B5$	NAME$
CUST.NAME$	CITY$	DESC$
CUST.CITY.STATE.ZIP$	DEBIT.CREDIT$	

The following are all unacceptable string variable names:

A	1B	COUNT%
CUST-NAME$	$NAME	AB

Strings have a value and a length. BASIC initializes all string variables to a length of zero--referred to as a null string--when a string variable is first referenced. During the execution of a program the length of a character string associated with a string variable can vary from zero to a limit of 255.

5.3 Array Variables

An array is a list or table of numeric, integer, or string variables with one or two subscripts. The subscript is a pointer to a specific location in a list or table in which a value is stored. You designate the pointer with either one or two subscripts enclosed by parentheses. When there are two subscripts they are separated by a comma. The value stored may be a numeric, integer, or string value, depending upon the array type.

To name an array start with a numeric, integer, or string variable name:

ITEMS	ITEMS%	ITEMS$

Then add the subscript reference:

ITEMS(4)	ITEMS%(2,10)	ITEMS$(15)

ITEMS(4) refers to the fifth value in the array ITEMS. It is the fifth value because the first value has a subscript of zero (a number base of 0). This may be changed by the OPTION statement.

ITEMS%(2,10) refers to the value "indexed" by row two, column ten in the table ITEMS%..

As mentioned, an array may have one or two subscripts. The number of subscripts is refered to as the number of dimensions of the array (see DIM statement). An array defined with one dimension must always be referenced with only one subscript. Likewise, an array defined with two dimensions must always be referenced with two subscripts.

Array names must be unique from variable names (the subscript references are not actually part of the name). This means that after the array ITEMS has been defined all references to a variable ITEMS are unacceptable because the name ITEMS is an array and must have subscripts. (The MAT statements are an exception to this because they only operate on arrays.) An attempt to use a variable name as an array and a non-array will result in an "Inconsistent usage" error.

Arrays are defined either explicitly with the DIM statement, or implicitly by using the array name in an assignment statement (LET) or as a term in an expression.

When an array is defined implicitly it is automatically dimensioned with an upper subscript of ten with one or two dimensions, depending upon the number of supscripts in the array reference. For example:

```
LET ITEMS(4) = 1234
```

will dimension the array ITEMS to have eleven elements with subscripts: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

```
LET ITEMS%(2,7) = 23
```

will dimension the array ITEMS% to have two dimensions with maximum subscripts of 10 in each dimension. This equates to 121 elements (OPTION BASE 0) or 100 elements (OPTION BASE 1).

If it is desired to have either fewer or more elements in an array you must use the DIM statement.

References to an array with a subscript greater than the size that the array was defined as will cause an error to occur (see ON ERROR GOTO statement and the appendix on error messages).

Most people find it inconvenient to work with a subscript number base of zero. For that reason the OPTION BASE 1 statement is provided. Refer to the OPTION statement for details on its use.

> Note: It is always a good practice to use the DIM statement to define the size of an array to avoid wasting storage space and to document the arrays and dimensions in use.

Array Example

As an example let the array ITEMS% be dimensioned to a size of 4 rows by 8 columns. To accomplish this you would use the statement: DIM ITEMS%(3,7) and the layout of the array would be:

		COLUMNS							
		0	1	2	3	4	5	6	7
R	0	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)
O	1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
W	2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
S	3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)

5.4 Functions

A function, in BASIC, is a special type of variable or constant. It is a predefined (or user defined) series of numeric and/or string operations.

A function name looks very much like an array name except that instead of one or two subscripts the function has zero or more "arguments". The arguments of a function are values that the function operates on or returns to the statement referencing it.

There are three types of functions:

- Intrinsic functions
- User defined functions
- USR functions

A function is used just like a variable or constant with one exception: a function cannot be assigned a value.

The following are all acceptable function names:

```
SQR(25)          Intrinsic - return square root of 25
INT(TOTAL)       Intrinsic - return integer value of TOTAL
FNA$(A1$,B2)     User defined function
USR(3,A$)        USR subroutine function
```

5.4.1 Intrinsic Functions

Intrinsic functions are functions that are an integral part of BASIC and need not be defined by the programmer.

The intrinsic functions provided with OASIS BASIC include functions to perform trigonometric operations, algebraic operations, general string operations, general numeric operations, logical operations, and screen control operations.

For a detailed description of the intrinsic functions refer to the chapter "Functions" later in this manual.

Intrinsic function names are all reserved words and as such, cannot be used as variable names. (See appendix on "Reserved Words".)

5.4.2 User Defined Functions

A user defined functions is one that must be defined by the programmer in each program.

A user defined function name always starts with the letters FN.

For a description of how to write a user defined function refer to the DEF statement.

When a reference is made to a user defined function name and that function is not defined with a DEF statement the reference will be interpreted as an array reference. This may cause an error when the function arguments are analyzed as subscripts.

5.4.3 USR Functions

A USR function is a call to a user written, assembly language, subroutine.

There can only be one USR function available at any one time, although it may have several "entry points".

Refer to the OPTION USR statement for details on the USR function. Refer to the OASIS MACRO Assembler Language Reference Manual for details on writing a USR function.

5.5 Expressions

Expressions are used extensively throughout this manual and within BASIC itself. Basically an expression is the specification of a series of operations to be performed on variables, constants, and functions, resulting in one value.

The use of an expression in BASIC is similar to expressions you use in your everyday work. For example, the term "work week" is used in estimating the time it takes to do a particular job. To determine the meaning of the term "work week" you normally multiply the number of hours a person works in a day by the number of days he works in a calendar week (normally 8 hours by 5 days). That is an example of an expression. Of course, in BASIC, you don't exactly use the same wording but it is quite similar:

```
LET WORK.WEEK = HOURS * DAYS
```

In BASIC there are several types of expressions:

- Arithmetic expressions
- String expressions
- Logical expressions
- Relational expressions

The type of an expression is determined by the type of operations it performs and the type of the constants, variables, or functions that it performs the operations on.

An expression can be as simple as a single constant or as complex as several hundred terms and operators.

The following are examples of expressions in BASIC:

```
2.345                          Arithmetic expression
A*SQR(GIRTH%)                  Arithmetic expression
NAME$&"abcdefg"                String expression
"Name: "&SPACE$(4)&NAME$       String expression
A OR B                         Logical expression
NOT TRUE%                      Logical expression
NAME1$ > NAME2$                Relational expression
CAT <= (BIRD AND DOG)          Relational expression with
                                logical subexpression
```

An expression is composed of terms (constants, variables, and/or functions) and operators (+, *, &, etc.). Operators are either binary operators (operate on two terms) or unary (operate on one term). An example of a binary operator is the multiplication operator (*). An example of a unary operator is the negative operator (-). Some operators can be either binary or unary such as the plus operator (+).

Expressions are frequently used in BASIC assignment statements (LET) but there are many other uses for expressions. The syntax of each of the statement descriptions specifies where an expression can be used and what type of expression is allowed.

Although there are four distinct types of expressions many expressions used in a BASIC program are generally a combination of two or more types of expressions.

5.5.1 Arithmetic Expressions

The arithmetic expression is the most common type of expression. An arithmetic expression has an arithmetic value (integer or floating point) and is defined as:

<arithmetic term> [<arithmetic operator> <arithmetic term>]

Arithmetic term

An arithmetic term may consist of any of the following:

```
Numeric constant
Integer constant
Numeric variable or array
Integer variable or array
Numeric function
Integer function
Logical expression
Relational expression
Arithmetic expression
```

Arithmetic operators

Operator	Function	English
	Exponentiation	raised to the power
*	Multiplication	times
/	Division	divided by
+	Addition (or unary positive)	plus
-	Subtraction (or unary negative)	minus

An arithmetic expression whose terms are mixed in type (both integer and floating point) yields a floating point value. An arithmetic expression whose terms are the same type (all integer or all floating point) yields a value of the same type.

You cannot place two arithmetic operators together unless the second operator is a unary minus or unary plus.

The following are examples of valid arithmetic expressions:

```
A%                                 Integer result
A%+23                              Integer result
SUB.TOTAL+CURRENT*UNIT.PRICE       Numeric result
ONE%*THREE                         Numeric result
+1/-4                              Numeric result
PI*RADIUS^+2                       Numeric result
3*4/(PI*R^2)                       Numeric result
```

Note that the last example uses parentheses. Parentheses may be used anytime to clarify the sequence of operations or to change the sequence (see section on

"Evaluating Expressions" below).

5.5.2 String Expressions

A string expression has a string value and operates only on string terms.

String term

A string term may consist of any of the following:

String constant
String variable or array
String function
String expression

String operator

A string operator may consist of:

Operator	Function	English
&	Concatenation	is concatenated with
[n:m]	Substring	from character n through m (unary operator)

The concatenation operator allows two strings to be joined together. Thus "ABCDEF"&"GHIJKLMN" produces ABCDEFGHIJKLMN. The concatenation operator always operates on two terms (binary operator).

The substring operator extracts characters from a string. The n in the operator represents the starting character position; the m in the operator represents the ending character position. The result of a substring operation is always a string of length m-n+1, even when one or both of n and m are greater than the current length of the string being operated on. The m value must be greater than or equal to the n value. The substring operator is a unary operator that operates on the preceding term, rather than the following term like other unary operators. The substring operator may be followed by the concatenation operator.

The following are examples of string expressions: (assume all string variables contain the constant "ABCDEFGH")

Expression	Result
NAME$	ABCDEFGH
"John Doe"	John Doe
A$&"Message"	ABCDEFGHMessage
CITY$&", "&ST$&" "&STR$(ZIP%)	ABCDEFGH, ABCDEFGH 12345
A$[2:4]	BCD
'"'&ALPHA$[4:9]&ALPHA$[11:12]&'"'	"DEFGH "
(A$&B$[5:9])[3:10]	CDEFGHEF

Note that in the last example parentheses were used. Parentheses are discussed in the section "Evaluating Expressions" below. In this example the parentheses are used to produce a "sub-expression" for the second substring operator.

The following are invalid string expressions:

A$/"ABCD"	Cannot include arithmetic operator
A$[4:1]	Invalid substring reference
A$>B$	Relational expression (see below)
A$&123	Cannot include arithmetic term
A$[2:3]&	Concatenation requires two terms

5.5.3 Logical Expressions

A logical expression operates on integer values and produces an integer value. A logical expression is defined as:

<arithmetic term> <logical operator> <arithmetic term>

Arithmetic term was defined above in the section "Arithmetic Expressions".

Logical operators

A logical operator is any of the following:

Operator	Function
NOT	Invert bits (on -> off; off -> on) in one term (unary)
AND	Tests for bit on in both terms
OR	Tests for bit on in either term
XOR	Tests for bit on in either but not both terms
IMP	Test first term--if bit on then bit bust be on in second te
EQV	Tests for equality--both bits on or both off

Logical expressions are comparisons between the corresponding "bits" of the two terms of the expression. A bit is a binary (either on or off) piece of information. An integer value is composed of sixteen bits. A decimal integer is expressed in bits by converting the number to base two notation and adding any leading binary zeros, if necessary. The following is a list of some equivalent values in decimal and binary:

Decimal	Binary bits
0	00000000 00000000
1	00000000 00000001
5	00000000 00000101
23	00000000 00010111
100	00000000 01100100
32767	01111111 11111111
-32767	10000000 00000000
-1	11111111 11111111

Note that a decimal zero has all zero bits and a decimal minus one has all one bits. This relationship between decimal and binary is used in the result of relational expressions, discussed in the following section.

The terms of a logical expression must be integers. When the terms are floating point in value BASIC will integerize them before the logical operation is performed.

Logical expressions are valid wherever arithmetic expressions are allowed in BASIC, however, both terms must be integers (floating point terms will automatically be "fixed").

The following tables are called truth tables. They show graphically the results of the logical operations for every possible combination of two bits.

Logical Truth Tables

NOT

A%	NOT A%
0	1
1	0

OR

A%	B%	A% OR B%
0	0	0
0	1	1
1	0	1
1	1	1

AND

A%	B%	A% AND B%
0	0	0
0	1	0
1	0	0
1	1	1

XOR

A%	B%	A% XOR B%
0	0	0
0	1	1
1	0	1
1	1	0

IMP

A%	B%	A% IMP B%
0	0	1
0	1	1
1	0	0
1	1	1

EQV

A%	B%	A% EQV B%
0	0	1
0	1	0
1	0	0
1	1	1

The following are examples of valid logical expressions:

```
NUM1% OR NUM2%
I% AND 23
I% AND (NUMBER XOR TOTAL) IMP TEST%
(A AND B) OR (A AND C)
STRING$ >= "A" AND STRING$ <= "Z"
```

Note that in the next to the last example parentheses were used. Parentheses are discussed in the section "Evaluating Expressions" below. In this example the parentheses are used to specify the sequence of evaluation.

The following are all unacceptable logical expressions:

```
STRING$ OR "HELP"          Must be arithmetic terms
NUM1% AND OR NUM2          Binary operators cannot be
                            adjacent
```

Logical expressions are normally used to evaluate terms that are the result of relational expressions (bits all on or all off); however, since the logical expression does compare all sixteen bits of each of the terms there are many other uses for logical expressions. One of the more common of these other uses is binary coded information or "bit swtiches".

Some examples will illustrate how the logical operators work on non-relational values:

```
15 AND 14
                    0000000000001111 (15)
                AND 0000000000001110 (14)
                    ----------------
                    0000000000001110 (14) (True)

10 OR 23
                    0000000000001010 (10)
                 OR 0000000000010111 (23)
                    ----------------
                    0000000000011111 (31) (True)

NOT 153
                NOT 0000000010011001 (153)
                    ----------------
                    1111111101100110 (-154) (True)

25 XOR 13
                    0000000000011001 (25)
                XOR 0000000000001101 (13)
                    ----------------
                    0000000000010100 (20) (True)

29 XOR 29
                    0000000000011101 (29)
                XOR 0000000000011101 (29)
                    ----------------
                    0000000000000000 (0) (False)

234 EQV 3429
                    0000000011101010 (234)
                EQV 0000110101100101 (3429)
                    ----------------
                    1111001001110000 (-3472) (True)

56 IMP 720
                    0000000000111000 (56)
                IMP 0000001011010000 (720)
                    ----------------
                    1111111111010111 (-41) (True)
```

As you can see there doesn't appear to be a relationship between the decimal terms and the decimal result of the expression; however, using the binary representations of the integers (as BASIC does) there is a definite, Boolean, relationship. This can be utilized to make an integer value contain sixteen, binary (on/off) switches. When using binary switches the logical expressions can be utilized to set or mask the number to expose the bit switch desired.

5.5.4 Relational Expressions

A relational expression operates on numeric or string terms and produces a sixteen (16) bit integer value of -1 (true - all bits on) or 0 (false - all bits off). A relational expression is defined as:

<arithmetic term> <relational operator> <arithmetic term>

or

<string term> <relational operator> <string term>

Arithmetic and string terms were defined above under the sections "Arithmetic Expressions" and "String Expressions" respectively.

Relational operators

A relational operator is any of the following:

Operator	Function
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
=	Equal to
<>	Greater than or less than (unequal to)

The following are all acceptable relational expressions:

```
STRING$ > "HELLO"          String relation
NUM1 <= NUM2               Numeric relation
NUMBER% <> 225*(5-ONE)     Numeric relation with
                            arithmetic sub-expression
539 = ONE                  Numeric relation (not assigment)
```

The following are all unacceptable relational expressions:

```
"Goodbye" <> 25            Can't mix string with numerics
NUM1 # NUM2                Invalid operator
```

5.5.5 Expression Evaluation

BASIC evaluates expressions according to operator precedence. Each arithmetic, string, logical, and relational operator joining an expression has a predetermined position in the hierarchy of operators. The operator's position tells BASIC when to evaluate the operator in relation to the other operators in the same expression.

Parentheses may be used to change the sequence of evaluation of an expression. Nested parentheses (one set of parentheses within another) may be used to cause the innermost subexpression to be evaluated first.

Parentheses may also be used as a documentation aid to clarify a complex expression.

The following table lists all of the expression operators in the hierarchy of evaluation.

Operator Precedence

Operator	Hierarchy
()	0
^ (exponentiation)	1
functions	2
substringing	2
+ (unary)	3
- (unary)	3
* (multiplication)	4
/ (division)	4
+ (addition)	5
- (subtraction)	5
& (concatenation)	5
> (greater than)	6
>= (greater or equal)	6
< (less than)	6
<= (less then or equal)	6
= (equal)	6
<> (unequal)	6
NOT	7
AND	8
OR	9
XOR	10
EQV	11
IMP	12

Notice that some operators have the same hierarchy number. This means that they are equivelent in precedence and will be evaluated in a left to right manner. This also applies to an expression with more than one occurence of the same operator.

As an example, consider the following expression:

```
A = 15^2+12^2-35*8
```

BASIC evaluates this expression in five, ordered steps:

```
1. 15^2 = 225       Exponentiation (left most)
2. 12^2 = 144       Exponentiation (next)
3. 35*8 = 280       Multiplication
4. 225+144 = 369    Addition
5. 369-280 = 89     Subtraction
   Result is 89
```

Arithmetic expressions with mixed arithmetic types (floating point and integer) will "float" all of the terms before expression evaluation.

As mentioned, parentheses can alter the sequence of evaluation (and possibly, the result). Consider the following, similar expressions and evaluations:

```
25^2+30^2/2                 (25^2+30^2)/2

1. 25^2 = 625               1. 25^2 = 625
2. 30^2 = 900               2. 30^2 = 900
3. 900/2 = 450              3. 625+900 = 1525
4. 625+450 = 1075           4. 1525/2 = 762.5
   Result is 1075              Result is 762.5
```

Note that in the above precedence table the relational operators have precedence over the logical operators.

Consider the following expression and evaluation:

```
"A">"B" OR "A"<="D"

1. "A">"B" = 0                  0000000000000000 (false)
2. "A"<="D" = -1                1111111111111111 (true)
3. 0 OR -1 = -1                 1111111111111111 (true)
```

CHAPTER 6

FORMATED OUTPUT

Sometimes the format of output is as important as the content. BASIC provides a means of controlling this format with the PRINT USING statement and the FORMAT$ function. Both the statement and the function allow you to control the appearance of data, thus enabling you to create formatted list, tables, reports, and forms.

The following example programs print a series of numbers. One program uses the PRINT statement and the other uses the PRINT USING statement.

```
0010 PRINT 1
0020 PRINT 10
0030 PRINT 123.5
0040 PRINT 100
0050 PRINT .23433
0060 PRINT 1000000
0070 PRINT -3

-RUN
 1
 10
 123.5
 100
 0.23433
 1000000
-3
```

```
0005 MASK$ = "#,#######.##-"
0010 PRINT USING MASK$,1
0020 PRINT USING MASK$,10
0030 PRINT USING MASK$,123.5
0040 PRINT USING MASK$,100
0050 PRINT USING MASK$,.23433
0060 PRINT USING MASK$,1000000
0070 PRINT USING MASK$,-3

-RUN
          1.00
         10.00
        123.50
        100.00
          0.23
1,000,000.00
          3.00-
```

As can be seen the PRINT statement left justifies numbers, performs no rounding, and indicates negative values with a leading, floating, minus sign; PRINT USING (and the FORMAT$ function) allows you to format numbers in several ways, making it easier to read and intepret the output.

There are several number formatting functions that the PRINT USING statement and the FORMAT$ function allows you to specify:

- Number of significant digits.
- Location of decimal point.
- Exponential format.
- Inclusion of special symbols (asterisk fill, dollar sign, commas, leading zeros).
- Alternate methods of indicating negative values (trailing sign, < >, trailing DB or CR).

There are also several string formatting functions that the PRINT USING statement allows you to specify:

- Number of characters.
- Left justified format.
- Right justified format.
- Center justified format.
- Extended format.

All of the formatting functions for the PRINT USING statement and FORMAT$ function are specified by using a mask that contains the formatting information. For details on the general syntax of the PRINT USING statement and FORMAT$ function refer to their respective sections.

PRINT USING and FORMAT Masks

PRINT USING and FORMAT$ masks are string expressions that contain formatting and non-formatting characters that control the format of the field output.

A PRINT USING mask may contain the format information for more than one field; a FORMAT$ mask contains the format informattion for only one, numeric field.

Non-formatting characters include any, and all, characters not specified here as formatting characters and act as literal information and field separators. These non-formatting characters will be included in the output field. Formating characters that operate in pairs (\\ $$ ** DB CR ^^ ^^^ **^ ^^^^) will act as non-formatting characters if they appear separately ($ * D B C R ^). Additionally, the formatting characters comma, period, minus, DB, CR, +, and > act as non-formatting characters if they appear separate from a numeric field specification.

For example:

```
0010 PRINT USING "Example 1: ##",1
0020 PRINT USING "This is a DB record 99",1
-RUN
Example 1:  1
This is a DB record 01
```

6.1 Numeric Field Masks

Numeric field masks may be used for both the PRINT USING statement and the FORMAT$ function. A numeric field mask requires a numeric value as input. When an attempt is made to use a numeric field mask with a string field the error "Invalid using" will occur.

The output of a numeric field specification mask will always be the same length as the length of the specification mask (unless there is insufficient space--see "Field Specification too Small"). If necessary, the number to be output is rounded by truncating digits to the right of the decimal point not specified in the mask or rounding the last digit specified to the right of the decimal point if the next digit was five or greater. When a number must be rounded to make it fit in the specified mask rounding will be performed on the absolute value of the number. For example:

```
0010 PRINT USING "##",1.4,1.5,1.6
RUN
 1
 2
 2
```

```
0010 PRINT USING "##",-1.4,-1.5,-1.6
RUN
-1
-2
-2
```

6.1.1 Specifying Number of Digits

All of the numeric, formatting characters are used to specify the total length of the output field; however, only the #, 9, comma, **, and $$ are used to specify the number of digits to be included. Conventionally, the # and/or 9 characters are used to specify the number of digits.

The 9 character reserves space for one digit and, if it appears before the decimal point specification, indicates that leading zeros are not to be suppressed. The 9 character cannot be used to format negative values.

The # character reserves space for one digit and indicates that leading zeros are to be suppressed.

The # and 9 characters may be mixed; however, if one or more 9 characters appear before the decimal point specification, leading zeros will not be suppressed.

When no sign specification is used and a negative value is output, a leading, floating, minus sign will be output, using one of the digit positions (as stated earlier, the output field will always be the same length as the mask field).

For example:

```
0010 PRINT USING "    #",1
0020 PRINT USING "   ##",1
0030 PRINT USING "#####",1
0040 PRINT USING "9####",1
0050 PRINT USING "#####"-1
0060 PRINT FORMAT$(23," ####")
-RUN
    1
    1
    1
00001
   -1
   23
```

```
0010 PRINT USING "    9",1
0020 PRINT USING "   99",1
0030 PRINT USING "99999",1
0040 PRINT USING "#9#9#",1
0050 PRINT USING "99999-",-1
0060 PRINT FORMAT$(23,"9####")
    1
   01
00001
00001
00001-
00023
```

6.1.2 Decimal Point Specification

You can specify the number of digits to the left and right of the decimal point by using a period embedded in the number field specification. The number of digits to the right of the decimal point specification will always be printed, even if zeros are required to do so.

If one or more digits are specified to the left of the decimal point there will always be at least one digit output, even if a zero is required to do so, unless

there is only one place specified, the number is negative and less than one, and there is no sign specification used, in which case the negative sign will be output immediately before the decimal point.

Specifying fewer places to the right of the decimal point than the number actually contains will cause rounding to occur to allow the number to fit. Specifying fewer places to the left will cause an error (see Specification too Small).

Only one decimal point may be specified in a numeric field mask. Specifying a second decimal point will indicate the end of the mask field and the start of another numeric field.

For example:

```
0010 PRINT USING "     .##",0
0020 PRINT USING "    #.##",1
0030 PRINT USING "   ##.##",1.2345
0040 PRINT USING "#####.####",1.24
-RUN
     .00
    1.00
    1.23
    1.2400
```

```
0010 PRINT USING "  .99",0
0020 PRINT USING " 9.99",1
0030 PRINT USING "99.99-",-.2
0040 PRINT USING " 9.999",-.234569
-RUN
  .00
 1.00
00.20-
%-.234569
```

6.1.3 Comma Specification

Commas may be inserted in the output field by using the comma character anywhere in the field, to the left of the decimal point specification, if used.

When the comma character is used the output field will be formatted with a comma appearing every third digit from the decimal point (or least significant digit if the decimal point specification is not used), working from right to left.

The comma character is also a digit specifier.

More than one comma may be specified for easier reading of the format mask: #,######### has the same effect as ###,###,### although the second form is more graphic in its meaning.

For example:

```
0010 PRINT USING "#,####",1
0020 PRINT USING "#,###########",1E^9
0030 PRINT USING "##,######.##",1234.56
-RUN
     1
1,000,000,000
    1,234.56
```

```
0010 PRINT USING "##,###",12345
0020 PRINT USING "#,###,###,###",1E^9
0030 PRINT USING "#,###,###.##,"1234.56
-RUN
12,345
1,000,000,000
    1,234.56
```

6.1.4 Dollar Field Specification

A number may be formatted with a dollar sign immediately before the most significant digit by using the floating dollar sign specification of two dollar sign characters together. (To format a number with a dollar sign before the field use a single dollar sign character and it will be treated as a non-formatting character.)

The double dollar sign characters must be at the start of the field (the asterisk fill specification, if used, must be before the floating dollar sign specification).

The double dollar sign characters indicate that a floating dollar sign is to be generated and one position is to be reserved for a digit.

If the number to be formatted is negative you must use the sign specification, otherwise a using error will occur.

The numeric field specification character 9 may not be used in a field with floating dollar sign specification. When it is it will be interpreted as the end of the field and the start of the next numeric field.

Extra dollar sign characters may be used instead of the # character. For instance, $$$$$$$$ is the same as $$#######.

For example:

```
0010 PRINT USING "$$###.##",12
0020 PRINT USING "$$###.##",1234
0030 PRINT USING "$$###.##-",-1234
-RUN
  $12.00
$1234.00
$1234.00-
```

```
0010 PRINT USING "      $$",1
0020 PRINT USING "$$,#####.##",12345
0030 PRINT USING "$$#######.##-",-12345
-RUN
     $1
 $12,345.00
 $12,345.00-
```

6.1.5 Asterisk Fill Specification

A number may be formatted with leading asterisk instead of leading zeros by using the asterisk fill specification of two asterisk characters.

The asterisk fill specification, if used, must appear at the very start of a numeric field specification.

The double asterisk characters indicate that any leading zeros are to be replaced with asterisks and that two positions are to be reserved for digits.

If the number to be formatted is negative you must use the sign specification, otherwise a using error will occur.

The numeric field specification character 9 may not be used in a field with asterisk fill specification. When it is it will be interpreted as the end of the field and the start of the next numeric field.

Extra asterisk characters may be used instead of the # character. For instance, ******** is the same as **######.

For example:

```
0010 PRINT USING "****.##",123
0020 PRINT USING "****.##-",-123
0030 PRINT USING "**$$$$-",-2
-RUN
*123.00
*123.00-
*****$2-
```

```
0010 PRINT USING "*****.##",123
0020 PRINT USING "**$$.##",1
0030 PRINT USING "*****",.5678
-RUN
*123.00
**$1.00
***1
```

6.1.6 Sign Specification

BASIC provides several methods of specifying how to print signed values. As stated above, when the mask field does not specify how to format a negative value, a leading, minus sign is generated. This is unacceptable in many cases and BASIC will not allow it if the format specification includes leading zeros (9), floating dollar sign ($$), or asterisk fill (**). In these situations you must use one of the sign specification characters.

All of the sign specification characters, when used, must appear at the end of format field (if they appear at the beginning or middle of a format field they will be treated as non-formatting characters or field separators, respectively).

Trailing Sign Specification

A plus sign character (+) at the end of a format specification indicates that the sign of the field (+ or -) is to be output at the end of the number.

Trailing Minus Sign Specification

A minus sign character (-) at the end of a format specification indicates that the sign of the field (-) is to be output at the end of the number if the value of the number is less than zero.

Trailing Debit Sign Specification

Debit specification characters (DB) appearing at the end of a format specification indicate that a literal DB is to be output at the end of the number if the value of the number is less than zero.

Trailing Credit Sign Specification

Credit specification characters (CR) appearing at the end of a format specification indicate that a literal CR is to be output at the end of the number if the value of the number is less than zero.

Angle Bracket Specification

An angle bracket character (>) at the end of a format specification indicates the the number is to be surrounded with angle brackets if the value of the number is less than zero.

Note that this specification is somewhat different from the other sign specifications in that not only is a character added at the end of the number output but also at the beginning of the number.

This sign specification may not be used with the numeric field specification characters 9, $$, or **.

Negative value specifications may be used with any of the other numeric field formatting characters with the exception of exponential field specification.

Examples:

```
0010 PRINT USING "####+",123
0020 PRINT USING "####-",123
0030 PRINT USING "####DB",123
0040 PRINT USING "####CR",123
0050 PRINT USING "####>",123
0060 PRINT USING "######.##>",12
-RUN
 123+
 123
 123
 123
 123
    12.00
```

```
0010 PRINT USING "####+",-123
0020 PRINT USING "####-",-123
0030 PRINT USING "####DB",-123
0040 PRINT USING "####CR",-123
0050 PRINT USING "####>",-123
0060 PRINT USING "######.##>",-12
-RUN
 123-
 123-
 123DB
 123CR
<123>
   <12.00>
```

6.1.7 Exponential Field Specification

BASIC normally prints a number in E format only when it is larger than 13 digits long, for example: 1234567890123 45 would be printed as 1.234567890123E+014. However, with PRINT USING or the FORMAT$ you can force a number to be output in E format. This is done with the exponential field specification: ^^^^^.

When a number is to be formatted in E format you cannot specify any other formatting characters other than the number of digits (#) or the decimal point position (.).

The exponential field specification, when used, must be at the end of the numeric field specification: ##.####^^^^^.

The exponential field specification may be used with fewer than five up-arrow characters when it is known that the exponent will fit in the smaller specification. For example:

```
^^^^^    allows for exponents from -126 to +126
^^^^     allows for exponents from -99 to +99
^^^      allows for exponents from -9 to +9
^^       allows for exponents from 0 to 9
```

For example:

```
0010 PRINT USING "#.##^^^^^",124
0020 PRINT USING "########^^^^^",123445
0030 PRINT USING "#####.####^^^^",12345678
0040 PRINT USING "#.#####^^",1234567
-RUN
1.24E+002
12344500E-002
12345.6780E+03
1.23456E6
```

6.1.8 Field Specification too Small

When a number field specification does not specify sufficient digit to allow the number to be output a percent symbol character (%) will be output followed by the number, unformatted.

This situation can happen for several reasons:

* Field isn't large enough: mask= ### number = 1234
* Field isn't large enough to include the commas specified: mask= #,### number= 12345
* Field isn't large enough to include floating dollar sign: mask= $$### number= 12345
* Field isn't large enough to include leading minus sign: mask= ### number= -123

In the following examples a double field mask is used to print two numbers, the first number won't fit in the first mask but the second, identical number will fit in the second mask.

```
0010 PRINT USING "### ####.##",1234,1234
0020 PRINT USING "#,### #####",12345,12345
0030 PRINT USING "$$### #####",12345,12345
0040 PRINT USING "**### #####",12345,12345
0050 PRINT USING "### ###-",-123,-123
0060 PRINT USING "#.##^^^^^ ##.##^^^^^",-1,-1
0070 PRINT USING "#.##^^ #.##^^^^",1E+12,1E+12
-RUN
% 1234  1234.00
% 12345  12345
% 12345  12345
% 12345  12345
%-123  123-
%-1  -1.00E+000
% 1000000000000  1.00E+12
```

6.2 String Field Masks

String field masks may only be used for the PRINT USING statement, not in the FORMAT$ function. A string field mask requires a string value as input. When an attempt is made to use a string field mask with a numeric value the error "[26] Invalid using" will occur.

The output of a string field specification mask will always be the same length as the length of the specification mask with one exception: extended fields. When a string value is longer than the string field mask BASIC will print as much of the string as will fit and truncate the remaining.

6.2.1 Single Character

You can specify that only the first character of the string value is to be printed by using the single quote character as a single character string mask field. Alternately the exclamation mark (!) may be used.

```
0010 PRINT USING "!","ABCDEFGH"
RUN
A
```

```
0020 PRINT USING "'","XYAX"
RUN
X
```

6.2.2 Left Justified Field

If you specify a left justified string field, BASIC prints the string starting at the left most position. If there are any unused places, BASIC prints spaces after the string. If there are more characters in the string value than in the string mask, BASIC truncates the string and does not print the excess characters.

To specify a left justified string field use the single quote lead in character (') followed by one or more L characters. The number of L characters (upper or lower case) plus the lead in quote specify the length of the left justified field.

Alternately you may use the back slant character to mark the beginning and end of the string mask. In this form spaces must be used between the two back slant characters. The number of spaces plus the two back slant characters specify the length of the left justified field to be printed.

With either method the minimum string length is two.

For example:

```
0010 PRINT USING "'L","ABCDEF"
0020 PRINT USING "'LLL","1234567"
0030 PRINT USING "\    \","ABC"
RUN
AB
1234
ABC
```

```
0010 PRINT USING "'LLLLL","1234567890"
0020 PRINT USING "'LLLLL","AB"
0030 PRINT USING "\   \","ABCD"
RUN
123456
AB
ABCD
```

6.2.3 Right Justified Field

If you specify a right justified string field, BASIC prints the string so that the last character of the string is in the right most place of the field. If there are any unused places before the string, BASIC prints spaces to fill the string. If there are more characters in the string value than in the string mask, BASIC truncates the string and does not print the excess characters.

To specify a right justified string field use the single quote lead in characters (') followed by one or more R characters. The number of R characters (upper or lower case) plus the lead in quote specify the length of the right justified field.

For example:

```
0010 PRINT USING "'RRRRRR","ABCD"
0020 PRINT USING "'RRRRRR","AB"
0030 PRINT USING "'RRRRRR","ABCDEF"
0040 PRINT USING "'RRRRRR","ABCDEFGHIJKLMNOP"
RUN
   ABCD
     AB
 ABCDEF
ABCDEFG
```

6.2.4 Center Justified Field

If you specify a centered field, BASIC prints the string so that the center of the string is in the center of the field. If the string cannot be exactly centered, such as a two character string in a five character field, BASIC prints the string one character off center to the left. If the length of the string is longer than the mask field the string will be truncated.

To specify a center justified string field use the single quote lead in character (') followed by one or more C characters. The number of C characters (upper or lower case) plus the lead in quote specify the length of the center justified field.

For example:

```
0010 PRINT USING "'CCCCCCCCCC","ABC"
0020 PRINT USING "'CCCCCCCCCC","ABCDEF"
0030 PRINT USING "'CCCCCCCCCC","A"
0040 PRINT USING "'CCCCCCCCCC","ABCDE"
0050 PRINT USING "'CCCCCCCCCC","ABCDEFGHIJKLMNOPQRSTUVWXYZ"
RUN
    ABC
  ABCDEF
     A
   ABCDE
ABCDEFGHIJK
```

6.2.5 Extended Field

The extended field is the only field that automatically prints the entire string. When you specify an extended field, BASIC left justifies the string as it does for a left justified field, but, if the string has more characters than there are places in the field, BASIC extends the field and prints the entire string. This extension may cause other items to be misaligned.

To specify an extended field use the single quote lead in character (') followed by one or more E characters. The number of E characters (upper or lower case) plus the lead in quote specify the minimum length of the extended field. The resulting output field will always be at least the length of the mask field.

For example:

```
0010 PRINT USING "'E-","ABCDEF"
0020 PRINT USING "'EEEE-","ABCDEF"
0030 PRINT USING "'EEEEEEEEEEEEEEEEEE-","ABCDEFGHIJKLMNOP"
RUN
ABCDEF-
ABCDEF-
ABCDEFGHIJKLMNOP    -
```

6.3 Multiple Fields In One Mask

The PRINT USING statement allows multiple fields to be specified in one mask. When this is done the values of the expressions in the PRINT USING statement are matched in a one to one relation with the fields in the mask. (The FORMAT$ function only allows one numeric field to be specified in the mask. A second field, if specified, will be used to mark the end of the mask.)

For example:

```
0010 PRINT USING "### #### #### ##%",1,2,3,4
0020 PRINT USING "999 9999 9999 99%",100,123,5,2
0030 PRINT USING "'RRRRRRRRRR ### 'E","ITEM",23,"THIS IS THE DESCRIPTION"
RUN
  1    2    3  4%
100 0123 0005 02%
       ITEM  23 THIS IS THE DESCRIPTION
```

As mentioned earlier, any non-formatting characters in the mask field are treated as literal characters to be included in the output:

```
0010 PRINT USING "ITEM 9999 Amount each: $$$$$$.##",23,15.40
RUN
ITEM 0023 Amount each:    $15.40
```

6.4 Re-using Mask Fields

The PRINT USING statement will re-use the mask field if there are more values specified as input than there are fields in the mask. BASIC will output a carriage return, line feed each time that the mask is re-used.

For example:

```
0010 PRINT USING "$$$$$$,$$$.##",1,23.4,34,234,5467.2,1235.924
RUN
         $1.00
        $23.40
        $34.00
       $234.00
    $5,467.20
    $1,235.92
```

6.5 Using Errors

A using error occurs (and a message is displayed) if:

- The format string is not a legal string expression.
- There are no valid fields in the format string.
- A string is printed in a numeric field.
- A number is printed in a string field.

PRINT USING and FORMAT$ Format Characters - Numeric Fields

Character	Function
9	Reserves place for one digit. Also specifies no zero suppression.
#	Reserves place for one digit, with leading zeros suppressed.
$$	Reserves place for one digit and floating dollar sign.
**	Causes leading asterisks to be printed instead of spaces. Also reserves place for two digits.
,	Causes a comma to be printed between every third digit starting from the decimal point and proceeding from right to left. Also reserves place for one digit.
.	Specifies location of decimal point.
-	Causes a trailing minus sign to be printed when number is negative.
+	Causes a trailing minus or plus sign to be printed depending upon the sign of the number
DB	Causes a trailing DB to be printed when number is negative.
CR	Causes a trailing CR to be printed when number is negative.
>	Causes a leading, floating < and a trailing > to be printed when number is negative.
^^	Causes the number to be printed in E format. Only allows for single digit, unsigned exponent.
^^^	Causes the number to be printed in E format. Only allows for single digit, signed exponent.
^^^^	Causes the number to be printed in E format. Only allows for double digit, signed exponent.
^^^^^	Causes the number to be printed in E format.

PRINT USING Format Characters - String Fields

Character	Function
!	Single character field printed.
\	Marks beginning or ending of a left justified field and reserves one place for a character.
'	Single character field printed or treated as the lead in character for following four format characters and reserves one place for a character.
L	Causes string to be left justified and reserves place for one character. Also lower case l.
R	Causes string to be right justified and reserves place for one character. Also lower case r.
C	Causes string to be center justified and reserves place for one character. Also lower case c.
E	Causes string to be left justified, reserves place for one character, and causes entire string to be printed. Also lower case e.

CHAPTER 7

USING FILES

BASIC supports file input and output to the on-line disk drives, console, printers, and other devices. Various file access methods are supported: SEQUENTIAL (one record after another from beginning of file); DIRECT (random by relative record number); INDEXED (random by key); KEYED (random by key).

Files have both an external name by which it is known within the system, and an internal file designator used within the BASIC program. For example, a file might exist on a disk, with the name INVEN.MASTER. This is the external name (i.e., INVEN.MASTER:A). In the BASIC program it might be opened on channel 1. This is done through the OPEN statement. All further references to the file in the program will be to #1 not to the file name of 'INVEN.MASTER:A'.

There are sixteen (16) channel numbers available to the user program, and all sixteen may be in use at one time. This means that there can be sixteen data files available for use at any one time in the BASIC program. Each open I/O channel requires buffer space and a small amount of space used for pointers, etc. The amount of buffer space needed varies, depending upon the device.

A seventeenth channel is always open to the CONSOLE. This channel is only accessed with INP and EOF functions, and the INPUT, LINPUT, and PRINT statements.

The sequence of statements in a BASIC program that uses a file is:

```
OPEN
INPUT, LINPUT, PRINT, READ, WRITE, etc.
CLOSE
```

OPEN This statement must be used before other file access statements to specify the file to be used, the internal channel to use for the file, the access mode and method, and various options that are to be used with the file.

INPUT,PRINT These statements perform the input and output to the file. They are performed as often as necessary to accomplish the function of the program. The specific statement to be used depends upon the access mode used in the OPEN statement and the file format.

CLOSE This statement is used last to designate that the operations to that file are complete.

7.1 Access Mode

There are three types of access modes that may be specified with the BASIC OPEN statement.

INPUT This mode indicates that the file is to be used for input operations only. When this mode is in effect BASIC will not allow output type operations to be performed on the file's I/O channel.

OUTPUT This mode indicates that the file is to be used for output operations only. BASIC will not allow input type operations to be performed on this file's I/O channel. This mode is normally used when a file is first being built or created or on output only devices like a printer.

UPDATE This mode allows both input and output operations to be performed on the file.

7.2 Access Methods - File Formats

The OPEN statement requires that you specify the access method of the file. This is the same as the file's format.

SEQUENTIAL Indicates that the records in the file are to be read or written sequentially, one after the other, starting at the beginning of the file. With this access method, to access any specific record, all records before that record must be accessed.

Records in this type of file are of variable length and the file does not have to be pre-allocated before it is used.

DIRECT Indicates that the records in the file are to be read or written randomly, by record number. This access method allows any record in the file to be accessed without accessing any other record in the file (i.e., directly).

This file format is quite useful for frequently accessed master files that

have sequentially numbered keys such as a customer file or a vendor file. Access to this type of file is fast, as the system can compute the address of the record on the disk without searching a separate index.

Direct files are only supported on disk devices, and must be created with the CSI command CREATE.

INDEXED Indicates that the records in the file are to be read or written randomly, by record key. This access, similar to DIRECT, allows any record in the file to be accessed without accessing any other record in the file; however, the record is accessed using a generic key, or name, of the record. This type of file is also maintained in alphabetic sequence by key and may be read in the sorted order.

Because of the necessity of keeping the index in sequence, updating this type of file is slower than using the direct or keyed file format.

Indexed files are only supported on disk devices, and must be created by the CSI command CREATE.

KEYED Indicates that the records in the file are to be read or written randomly, by record key. This access is identical to INDEXED except that the file is not maintained or accessible in any sorted order.

7.3 Record Allocation Requirements

Since indexed, direct, and keyed files must be preallocated by the user before the BASIC program can access them it is necessary for the user to calculate the maximum record size required for each file. To do this the user must determine the field types to be written to the file.

For each string field in a record the user must allocate space for the length of the longest field plus 2.

For each floating point field in a record the user must allocate space for 9 positions.

For each integer field in a record the user must allocate space for 3 positions.

Thus the record size for the following direct file must be 32:

```
WRITE #1,N:"RECORD",1,2,A,B
```

7.4 Multi-User File Protections

A BASIC program run on a multi-user OASIS system will operate the same as on a single user OASIS system, except that file contention may occur. This means that two users may attempt to access the same file or the same record in a file at the same time. This situation may, or may not be allowed, depending on the file protections used by the two programs.

A program that does extensive input and output to a file should lock the entire file from other user's use. This is done by specifying the LOCK option in the OPEN statement.

If a file is not locked in its entirety other users may access the file (unless the other user attempts to lock the entire file which would not be allowed).

When a file is opened for INPUT or OUTPUT no record locking will be performed and it is possible that a record read by your program might be updated by another user's program without your program's knowledge. This could result in errors in the file.

When programming in a multi-user system the programmer must always ask the question: What happens if another user wants this record? and program accordingly.

(This page intentionally left blank)

CHAPTER 8

COMMANDS

BASIC commands are used to enter, change, and debug programs. They only may be used in the command mode. Command mode is when BASIC prompt character is displayed (-).

BASIC command functions may be divided into four categories:

A. General

HELP - Display list of commands available.

LENGTH - Display current memory utilization of program.

NAME - Display or change name of program in memory.

NEW - Initialize BASIC work area, new program.

B. Editing

AUTO - Automatic line number prompting for new line entry.

BOTTOM - Position to the last line in the program.

CHANGE - Change string in one or more lines of code.

DELETE - Remove line(s) of code from program.

DOWN - List next line.

INDENT - Perform standard program indentation.

LIST - List one or more lines of program.

LOCATE - Locate line containing string.

LPLIST - List one or more lines of program on printer.

LPXREF - List cross reference table on printer.

MODIFY - Character by character change of one or more lines.

RENUMBER - Renumber all or part of program.

TOP - Position to the first line in the program.

UP - List prior line.

XREF - List cross reference table on terminal.

C. Disk programs

LOAD - Retrieve program from disk.

RUN - Execute program from disk or already in memory.

SAVE - Save current program on disk.

D. Debugging

BREAK - Specify condition to break on.

CONTINUE - Resume execution.

STEP - Execute next statement and stops.

TRACE - Display line numbers executed and optionally variables changed.

UNTRACE - Discontinue trace mode.

UNBREAK - Remove one or all breakpoints set.

VARS - Display contents of all variables defined.

8.1 AUTO Command

1 AUTO

2 AUTO <start>

3 AUTO <start> <increment>

Where:

<start> ::= <line number>
<increment> ::= <line increment value>

Purpose:

The AUTO command allows you to enter new lines to the program with automatic line numbering.

Comment:

The AUTO command cannot be used if a program in memory is read protected (see LOAD command).

The AUTO command is intended to be used for creating new programs or adding new sections to an existing program in memory.

The <increment> value, when specified, sets the current increment value for this AUTO and subsequent executions of the AUTO command.

When the AUTO command is executed BASIC will display the current line number plus the current increment on the console (or the <start line number>, when specified) followed by a space. You may then enter a program line. After a program line has been entered and terminated by a carriage return, the line number is incremented by the current increment and the process is repeated.

To terminate the line input process enter a carriage return when BASIC prompts you with the line number. No blank line will be added to the program. Lines entered with this command cannot replace any line in the program with the same line number nor can it be used to add multiple lines that merge around existing lines. In order to add lines that merge around exising lines you must enter them one at a time.

Examples:

```
-LIST
  10 INPUT "RADIUS OF CIRCLE",RR
  20 PRINT "DIAMETER =";2*R
  30 PRINT "AREA =";PI*R^2
  40 PRINT "CIRCUMFERENCE =";2*PI*R
-DELETE 10 20
TOF:
-AUTO
  10 PRINT "HELLO"
  20 LET R=55
AUTO cannot replace or merge lines

-LIST
  10 PRINT "HELLO"
  20 LET R=55
  30 PRINT "AREA =";PI*R^2
  40 PRINT "CIRCUMFERENCE =";2*PI*R
-AUTO 50,3
  50 PRINT "AGAIN";\INPUT Y$
  53 IF Y$="Y" THEN 20
  56 END
  59
```

8.2 Bottom Command

BOTTOM

Purpose:

The bottom command positions to the last line in the program and displays that line of code.

Comment:

The bottom command cannot be used if the program in memory is read protected (see LOAD command).

Examples:

```
Stop at line 0020
-BO
 999      END
```

8.3 BREAK Command

```
1 BREAK
2 BREAK [AT] <line reference>
3 BREAK [AT] <line reference> [AFTER] <count>
4 BREAK [ON] <variable>
5 BREAK [ON] <variable> CHANGE
6 BREAK [ON] <variable> AFTER <count>
7 BREAK [ON] <variable> CHANGE AFTER <count>
8 BREAK [ON] <variable> <relation> <value>
Where:
     <line reference> ::= <line number>
                          <line label>
     <relation> ::= <relational operator>
     <value> ::= <numeric literal>
                 <quoted string literal>
                 <numeric variable>
                 <string variable>
See also: STEP, TRACE, UNBREAK, UNTRACE, and VARS commands
```

Purpose:

The BREAK command provides the capability of dynamic debugging of the BASIC program.

Comment:

<count> is a numeric value referring to the number of times that the specified break condition is to occur before a break is actually performed.

<variable> is a simple numeric variable, not a subscripted variable. An array name is acceptable.

Format 1 of the BREAK command will display the current break table.

Format 2 will cause a break to occur at the next execution of the statement on the line referenced, before the statement is executed.

Format 3 will cause a break to occur at the <count> execution of the statement on the line referenced, before the statement is executed.

Format 4 will cause a break to occur the next time that the <variable> is used, after the statement using the variable is executed.

Format 5 will cause a break to occur the next time that the <variable> is changed by a statement, after the statement changing the variable is executed.

Format 6 will cause a break to occur after the <variable> is referenced <count> times, after the statement referencing the variable the <count> time is executed.

Format 7 will cause a break to occur after the <variable> is changed <count> times, after the statement changing the variable the <count> time is executed.

Format 8 will cause a break to occur when the relationship is true, after the statement causing the relationship to become true is executed.

The BREAK command may be abbreviated to the letter B.

When a break occurs, execution of the program stops and the message "Break at ...." or "Break on ...." is displayed. Control returns to the command mode. When a break occurs on a variable reference or change the statement causing the break will be completly executed. Executing a CONTINUE command will cause the statement

following to be executed.

Only one break will be set for a specific variable or line at one time. When multiple break points are attempted to be set for a variable or a line only the last one specified will be in effect.

Note: Break points are only cleared by the UNBREAK, NEW, and LOAD commands. During execution, if a different program is brought into memory the old break points will still exist. The RENUMBER command does not change the line numbers specified in any break points.

==========

Example:

```
-LIST
  10 FOR I%=1 TO 4
  20       PRINT I%
  30       GOSUB SUM
  40       NEXT
  50 GOTO 9999
  60 SUM: TOTAL% = TOTAL%+I%  RETURN
9999 END

-BREAK AT SUM
-BREAK ON I% CHANGE AFTER 4
-BREAK
Break at SUM
Break on I% changed after 4
-RUN
Break at line 60
-VARS TOTAL%
TOTAL% = 0
-VARS I%
I% = 1
-UNBREAK AT SUM
-CONTINUE
Break on I% at line 40
-VARS I%,TOTAL%
I% = 4
TOTAL% = 6
-
```

==========

8.4 CHANGE Command

1 CHANGE

2 CHANGE <char><from string><char><to string><char>

3 CHANGE <char><from string><char><to string><char><range>

Where:

<char> ::= <delimiting character>
<from string> ::= <string>
<to string> ::= <string>
<range> ::= <line number>[<line number>]

See also: MODIFY command

Purpose:

The CHANGE command allows you to make a change to an existing line, or lines, of code without re-entering the entire line.

Comment:

The CHANGE command cannot be used if a program in memory is read protected (see LOAD command).

Format 1 of the CHANGE command will execute the last executed CHANGE command on the current line.

Format 2 of the CHANGE command will change all occurrences of the <from string> on the current line to the <to string>.

Format 3 of the CHANGE command will change all occurrences of the <from string> on each line of the lines within <range> to the <to string>.

The <from string> and <to string> must be delimited by the same character, similar to the CHANGE command in the system editor. The delimiters must be quotation marks if you wish to change from or to a mixed or lower case string. You may not change from a mixed case string to a mixed case string. To do that you must use the MODIFY command.

Each time that the CHANGE command actually makes a change on a line the line will be displayed with the change made.

Note: To change only one occurrence on a line use the MODIFY command.

Examples:

```
-LIST
  10 INPUT "Item 1",R
  20 PRINT R
  30 INPUT "Item 2",R1
  40 PRINT R1
-CHANGE /INPUT/LINPUT/ 10 40
  10 LINPUT "Item 1",R
  30 LINPUT "Item 2",R1
-CHANGE "Item"VALUE" 10
  10 LINPUT "VALUE 1",R
-
```

Explanation:

8.5 CONTINUE Command

1 CONTINUE

Purpose:

The CONTINUE command allows you to resume execution of a program that was interrupted.

Comment:

The CONTINUE command, when executed, will continue the execution of a program whose execution was interrupted by a STOP statement, an error, or entry of the Program Cancel-Key.

When a program has a normal exit, i.e., execution of the END statement, the CONTINUE command has no effect.

The CONTINUE command is a valuable debugging aid. If a "bug" is suspected in a portion of a program, STOP statements may be inserted at strategic positions of the program. When the STOP is executed, you may use the commands to examine variables and/or change statements in the program and continue execution. If an error occurs, you may examine the suspected statement and change it as required and continue execution.

When an error occurs, a CONTINUE command will re-execute the line that contained the statement that was interrupted. If the error occurs in a multi-statement line, the CONTINUE command re-executes the entire line.

If a STOP occurs, a CONTINUE command will execute the statement following the STOP statement, even if that statement is on the same line.

Note: Executing an immediate instruction after a stop or error has occurred will prevent you from using the CONTINUE command. An immediate instruction, when executed, causes the line pointer to be lost.

8.6 DELETE Command

1 DELETE

2 DELETE <range>

Where:

<range> ::= <line number> <line number>
<line number>,<line number>

Purpose:

The DELETE command allows the user to remove a line or group of lines from the program.

Comment:

Format 1 of the DELETE command removes the current line from the program in memory.

Format 2 of the DELETE command removes all lines from the program in memory whose line numbers are included in the range specified.

The DELETE command cannot be used if the program in memory is read protected (see LOAD command).

The restrictions for first and last line numbers as described for LIST apply to the DELETE command; however, the DELETE command must have at least one operand.

Using the first example in Appendix G as the program in memory:

Examples:	Explanation:
-DELETE 40	Line 40 is removed from the program.
-DEL 15,45	Lines 20 and 30 are removed from the program.
-LIST 10 INPUT "RADIUS OF CIRCLE",R 50 END	

Incorrect Examples:	Explanation:
-DELETE 50,20 Invalid command syntax	Last line number must be greater than or equal to first line number.

8.7 Down Command

<line feed>

Purpose:

The down command advances and displays the next line of source code.

Comment:

The down command cannot be used if the program in memory is read protected (see LOAD command).

When the down command is entered the current line pointer is adjusted one line forward and the current line is displayed.

Attempting to executed the down command when the current line is at the last line of the program the message EOF: is displayed indicating that you are at the end of file.

8.8 HELP Command

1 HELP

Purpose:

The HELP command displays the commands available to the operator in BASIC.

Comment:

When the HELP command is executed the help message of command names and general syntax is displayed on the screen, one page at a time.

Example:

```
-HELP
AUTO [<start>[,<incr>]]
BOTTOM
BREAK [AT <line> [AFTER <count>]]
BREAK [ON <var> [CHANGE] [AFTER <count>]]
BREAK [ON <var <relat> <value>]
  .
  .
  .
```

8.9 INDENT Command

1 INDENT [<indent value>]

Purpose:

The INDENT command provides an easy and consistent method of performing program line indentation for documentation purposes.

Comment:

When the INDENT command is executed the program currently in memory is modified by stripping all current line indentation and performing new indentation according to a set of rules:

- Indent level initially set to <indent value> or, when not specified, to the default value of 5.
- The statements CASE, CEND, ELSE, IFEND, REM, and THEN cause the indent level to be adjusted -<indent value> before the statement.
- The statements CASE, multiline DEF, ELSE, FOR, REM, THEN, WHILE, and line label cause the indent level to be adjusted +<indent value> after the statement or label.
- The statements CEND, FNEND, IFEND, NEXT, and WEND cause the indent level to be adjusted -<indent value> after the statement.
- The statements IF and SELECT cause the indent level to be adjusted +2*<indent value> after the statement.
- All other statements perform no adjustment on the indent level.

Example:

```
-LIST
  10            REM This is a comment
  20     FOR I=1 TO 10
  30     PRINT I
  40     NEXT I
  50  SELECT A
  60           CASE 1  RETURN
  70         CASE 2  STOP
  80           CEND
  90    REM This is a subroutine
 100   RETURN
-INDENT
-LIST
  10 REM This is a comment
  20      FOR I=1 TO 10
  30           PRINT I
  40           NEXT I
  50      SELECT A
  60           CASE 1  RETURN
  70           CASE 2  STOP
  80           CEND
  90 REM This is a subroutine
 100      RETURN
-
```

8.10 LENGTH Command

LENGTH

Purpose:

The LENGTH command allows the user to determine the current memory utilization.

Comment:

The LENGTH command displays thirteen quantities:

* Length of source program in bytes.
* Memory space used by symbol table, numeric and integer variables, in bytes.
* Memory space used by string variable storage area in bytes.
* Memory space used by subroutines in process.
* Memory space used by FOR/NEXT loops in process.
* Memory space used by SELECT/CASE/CEND structures in process.
* Memory space used by WHILE/WEND structures in process.
* Memory space used by debugging break points.
* Memory space used by I/O channels used (266 bytes per channel).
* Memory space available for program and work usage. The expression analyzer requires about 512 bytes of this area during execution.
* USR program name, if loaded.
* USR program length, if loaded.
* USR program load address, in hexadecimal, if loaded.

Some programs may use all of memory. This command informs the user how much memory is available for modifications, how much memory was made available by modifications (by using the length command before and after).

Examples:

```
-LENGTH
Source: 3110                    Length of source program code.
Symbol: 178                     Symbols and numeric variable values.
String: 236                     String variable values.
GOSUB: 0                        No open subroutines
FOR/NEXT: 0                     No open FOR/NEXT loops.
CASE: 0                         No open SELECT/CASE/CEND structures.
WHILE: 0                        No open WHILE/WEND structures.
Debug: 4                        One breakpoint set.
Buffers: 532                    Two I/O channels defined.
Free: 19193

-OPTION USR "PRINT1"            Load USR program named PRINT1
-LEN
Source: 3110
symbol: 178
String: 236
GOSUB: 0
FOR/NEXT: 0
CASE: 0
WHILE: 0
Debug: 4
Buffers: 532
Free: 18784

USR NAME: PRINT1
Length: 409
Addr: F6A2
```

8.11 LIST Command

```
1 LIST

2 LIST <range>

3 <line number>

4 <carriage return>

Where:

     <range> ::= <line number>
                 <line number> <line number>
                 <line number>,<line number>

See also: Down, LPLIST, LPXREF, and XREF commands
```

Purpose:

The LIST command allows you to display a line, or group of lines, of the program.

Comment:

The LIST command cannot be used if the program in memory is read protected (see LOAD command).

Format 1 of the LIST command will list the entire program.

Format 2 of the LIST command with one line number will list only that line, if it exists.

When a program is in memory you may list a line by entering its line number (format 3), followed by a carriage return. This not only causes the specified line to be displayed but assigns that line number to an internal line display pointer. Entering a line feed character causes the line display pointer to be advanced to the next line and causes that line to be displayed. This provides an easy means of stepping through the display of a program. Additionally, the line display pointer is affected by an error during execution. When an error is detected and a message displayed on the console, you need only enter a carriage return to cause the error line to be displayed.

A carriage return only entry (format 4) will act in one of two ways: when it is the first entry of the carriage return the current line will be displayed; subsequent entries of a carriage return only will cause the internal line display pointer to be incremented causing the next program line to be displayed, acting like a line feed entry.

Format 2 of the LIST command with two line numbers will list all lines within the range of the operands, inclusive. The beginning and ending line numbers need not be line numbers that exist in the program. The last line number must be greater than or equal to the first line number.

When more lines are specified to be displayed than will fit on the console at one time and the console screen wait is enabled (see System Control Keys in the OASIS System Reference Manual), BASIC will display one page of the program, display a circumflex (^) at the lower left side of the page, and wait for the operator to respond. A response of any character will cause BASIC to display the next page of the program, if included in the line number range. The Program Cancel-key will cause the listing to be terminated immediately.

Examples:	Explanation:
-LIST 10 INPUT "RADIUS OF CIRCLE",R 20 PRINT "DIAMETER =";2*R 30 PRINT "AREA =";PI*R^2 40 PRINT "CIRCUMFERENCE =";2*PI*R 50 END	Entire program is listed on terminal.
-LIST 20 20 PRINT "DIAMETER =";2*R	Line 20 is listed.
-LIST 0,15 10 INPUT "RADIUS OF CIRCLE",R	Lines 0 through 15, inclusive, are listed.

Incorrect Examples:	Explanation:
-LIST 15	Since there is no line 15 nothing will be listed.
-LIST 20,10	Last line number must be greater than or equal to first line number.

8.12 LOAD Command

1 LOAD <program name>

Where:

```
<program name> ::= [<file name>][.<file type>][:<file disk>]
<file type> ::= BASIC
                BASICOBJ
```

See also: RUN command

Purpose:

The LOAD command allows the user to retrieve a program previously saved on disk.

Comment:

A program name must be specified but the program file type is optional. The program file type, if specified, may only be BASICOBJ or BASIC--no other program types are allowed.

The program file type defaults to BASICOBJ and BASIC:

When no file type is specified then a search is made for a program with the file type BASICOBJ. If one is found it is loaded.

If a program with a file type of BASICOBJ is not found then a search is made for a program with the file type of BASIC. If one is found then it is loaded with syntax analysis of each and every line.

It is much faster to load programs saved with a file type of BASICOBJ because no syntax analysis is performed.

When no program disk is specified the search for the program includes all attached disk drives.

If the specified program is not found then the error message "File not found" is displayed and no program is loaded. However, any program that was in memory before will have been erased.

The LOAD command can load a read protected program file; however, most other commands will not operate if the program in memory is from a read protected file. Specifically, the following commands will inform you that they cannot be used when the program is read protected: AUTO, BOTTOM, CHANGE, DELETE, INDENT, LIST, LOCATE, LPLIST, LPXREF, MODIFY, NAME, RENUMBER, SAVE, TOP, XREF, carriage return, line feed, up-arrow.

Examples:	Explanation:
-LOAD TEST	The program named TEST.BASICOBJ or TEST.BASIC will be located and loaded into memory.
-LOAD TEST:S	The program named TEST.BASICOBJ:S, or TEST.BASIC:S will be located and loaded into memory.

Incorrect examples:	Explanation:
-LOAD	Program name must be specified.
-LOAD TEST:T	Invalid unit specified.
-LOAD PROGRAM.TEST	Invalid file type.

8.13 LOCATE Command

```
1 LOCATE

2 LOCATE <string>

3 LOCATE <string> <range>

Where:

     <string> ::= <delimited string>
     <range> ::= <line number>
                 <line number> <line number>
                 <line number>,<line number>
```

Purpose:

The LOCATE command allows you to quickly find a line of the program that contains a specified sequence of characters.

Comment:

The LOCATE command cannot be used if the program in memory is read protected (see LOAD command).

The LOCATE command searches the program in the specified range of line numbers for the sequence of characters specified.

A LOCATE command with no arguments (format 1) will cause a LOCATE to be performed using the string specified in the last LOCATE or CHANGE, from the current line to the end of the program.

Format 2 of the LOCATE command causes the program to be searched from the line after the current line to the end of the program.

Format 3 of the LOCATE command with only one line number specified causes the program to be searched from the line specified to the end of the program.

Format 3 of the LOCATE command with two line numbers specified causes the program to be searched only within the range indicated.

If the sequence of characters is found the line containing them will be displayed and the current line pointer will be positioned at that line.

If the sequence of characters is not found nothing will be displayed and the current line pointer will not be changed.

The search is performed independent of the case mode of the characters in the program.

Example:

```
0010 FOR I=1 TO 20
0020 PRINT "Now is the time for all good men to come to the aid of"
0030 PRINT "country."
0040 NEXT I

-LOCATE "y" 10 40
0030 PRINT "country."
-LOCATE /T/
0040 NEXT I
-LOCATE /I/ 10 40
0010 FOR I=1 TO 20
-LOCATE
0020 PRINT "Now is the time for all good men to come to the aid of"
-
```

8.14 LPLIST Command

1 LPLIST

2 LP<n>LIST

Where:

<n> ::= 1
2
3
4

See also: Down, and LIST commands

Purpose:

The LPLIST command allows the user to list the current program on the list device (usually the line printer).

Comment:

The LPLIST command cannot be used if the program in memory is read protected (see LOAD command).

The LPLIST command functions identically to the LIST command except the output is placed on the listing device (PRINTER1) instead of the console and no line number range is allowed.

The alternate form of the command (LP<n>LIST) specifies that one of the alternate listing devices is to be used (PRINTER1, PRINTER2, PRINTER3, or PRINTER4), if attached.

8.15 LPXREF Command

```
1 LPXREF

2 LP<n>XREF

Where:

     <n> ::= 1
             2
             3
             4

See also: XREF command
```

Purpose:

The LPXREF command produces a listing of the program followed by a cross reference listing of the source program in memory on a printer.

Comment:

The LPXREF command cannot be used if the program in memory is read protected (see LOAD command).

The LPXREF command functions identically to the XREF command except the output is placed on the listing device (PRINTER1) instead of the console.

The alternate form of the command (format 2) specifies that one of the alternate listing devices is to be used (PRINTER1, PRINTER2, PRINTER3 or PRINTER4), if attached.

8.16 MODIFY Command

```
1 MODIFY

2 MODIFY <range>

Where:

     <range> ::= <line number>
                 <line number> <line number>
                 <line number>,<line number>

See also: CHANGE command
```

Purpose:

The MODIFY command allows you to make changes to a line or lines of code without re-entering the entire line.

Comment:

The MODIFY command cannot be used if the program in memory is read protected (see LOAD command).

The MODIFY command operates very similar to the MODIFY command in the OASIS system EDIT program.

When no <range> is specified the current line displayed and you are allowed to modify it. After finishing the modification of that line control returns to the command mode of BASIC.

When <range> is specified the first line included in the range of lines specified is displayed and you are allowed to modify it. After finishing the modification of that line the next line included in the range of lines specifies is displayed, etc., until the last line in the range is modified, at which point control returns to the command mode.

While in the modify mode of BASIC there is a certain set of sub-commands available to facilitate modification of each line:

I Allows you to insert characters at the current cursor position. All characters typed after the I has been entered are added to the line before the current character. As each character is added to the line the remainder of the line is re-displayed.

To exit from the insert character command type a carriage return.

While in the insert character command you may backup one character position by typing the RUBOUT key. This backs the cursor up one position and deletes that character. It is possible to backspace past the position that the insert command was given.

D Allows you to delete the current character from the line. Every time a D is typed the current character is deleted from the line and the character is erased from the screen.

R Allows you to replace characters in the line. All characters typed after the R has been entered will replace the characters in the line.

To exit from the replace character command type a carriage return.

While in the replace character command you may backup one character position by typing the RUBOUT key. This backs the cursor up one position without deleting that character. It is possible to backspace past the position that the replace command was given.

<sp> Allows you to advance the current character one position to the right. You may not advance past the end of the line, however, you may insert new characters at the end of the line or replace characters at the end of the line.

The right arrow key has the same effect as the space character.

F Allows you to advance the current character pointer to a specified character. The F character is followed by the character to find. When the second character is entered the cursor is advanced to the next occurrence of that character in the line.

U Allows you to convert characters to their upper case value. When the U is entered the current character is converted and re-displayed in its upper case form, and the cursor is advanced to the next character.

L Allows you to convert characters to their lower case value. When the L is entered the current character is converted and re-displayed in its lower case form, and the cursor is advanced to the next character. This command is only effective within a quoted string literal or a remark statement.

<RUB> Allows you to backspace the current character one position to the left. You may not advance past the beginning of the line.

The left arrow and CTRL/H have the same effect as the RUBOUT key.

B Allows you to quickly position to the beginning of the line.

E Allows you to quickly position to the end of the line.

<CR> Terminates the modification of the line.

ESC,C Terminates the modification of the line and restores the line to its original, unmodified, contents.

Due to the graphic and character by character nature of the modify command no example will be given here. Instead it is suggested that you experiment with it.

8.17 NAME Command

```
1 NAME

2 NAME <program name>

Where:

     <program name> ::= [<file name>][.<file type>[:<file disk>]]
     <file type> ::= BASIC
                     BASICOBJ

See also: SAVE command
```

Purpose:

The NAME command allows you to change the name of the program in memory.

Comment:

The NAME command operates in two modes: display current name (format 1); change current name (format 2).

Format 1 of the NAME command causes the current program name, type, and disk to be displayed.

Format 2 of the NAME command causes the current program name to be changed to the name specified. If the program type is omitted the program type will be changed to BASICOBJ. If the program disk is omitted the current program disk will be retained.

The file type of a read protected program should not be changed from BASICOBJ.

Note: The OASIS command RENAME should not be used to change the file type of a BASIC program due to unpredictable results.

Example:

```
-NAME TEST.BASIC:A
-NAME
TEST.BASIC:A
-NAME TESTIT
-NAME
TESTIT.BASICOBJ:A
```

8.18 NEW Command

1 NEW

Purpose:

The NEW command allows the user to enter a new program.

Comment:

The NEW command effectively clears memory. In actuality all of the BASIC pointers are reset to indicate that there is no program in memory. All of the BASIC work area is available for use by the new program to be entered. Additionally the name of the program is cleared.

A NEW command is executed automatically when BASIC is loaded and executed by the Operating System.

The NEW command is the only method of unloading a USR program without exiting BASIC.

Specifically, the NEW command performs the following actions:

* All files are closed.
* All file buffers are deleted from memory.
* The current program and program name are erased.
* Any USR program is erased and memory restored.
* All variables, constants, and internal tables are initialized.

Examples:

```
-NEW                                    Memory is initialized.
```

Incorrect examples:

```
-NEW TEST1                              No operand is allowed.
```

8.19 QUIT Command

```
1 QUIT
2 QUIT <string literal>
3 QUIT <numeric literal>
```

Purpose:

The QUIT command allows the user to exit from the BASIC environment.

Comment:

When the QUIT command is executed all open I/O channels are closed.

The QUIT command always exits from BASIC. If BASIC was invoked by a keyboard command then control is returned to the Command String Interpreter environment. If BASIC was invoked by an EXECutive procedure then control is returned to the EXECutive procedure that called it. The EXEC resumes control with the statement that followed the BASIC command. In either case the return code is set to zero.

To exit BASIC without returning control directly to the environment that it was invoked from one of the optional literals is specified.

A numeric value indicates the value that the return code is to be set to. This return code may then be examined by the EXEC that invoked BASIC. If BASIC was not invoked by an EXEC then setting the return code will have no usable effect.

A string value indicates a CSI command to be executed. The value must specify the command name and all arguments and options desired. After the command has completed execution the return code is set by that command. If BASIC was invoked by an EXECutive procedure and a string value is specified with the QUIT command, control will return to the EXEC program after the CSI command has completed execution.

When the first character of the string value is the character ">" the string will be displayed on the console terminal, just as if it had been entered from the keyboard.

Examples:	Explanation:
-QUIT	Control returns to the environment from which BASIC was invoked.
-QUIT LIST DAILY REGISTER	BASIC is exited and the file named DAILY REGISTER is listed on the console.

8.20 RENUMBER Command

```
1 RENUMBER
2 RENUMBER <first>
3 RENUMBER <first><char><increment>
4 RENUMBER <first><char><increment><char><start>
5 RENUMBER <first><char><increment><char><start><char><end>
Where:
     <char> ::= <space>
                <comma>
     <first> ::= <line number>
     <increment> ::= <line number>
     <start> ::= <line number>
     <end> ::= <line number>
```

Purpose:

The RENUMBER command allows you to resequence all or part of the program in memory.

Comment:

Format 1 of the RENUMBER command will resequence all of the program in memory, from the beginning of the program through the end of the program. The resulting program will have its first line numbered using the default increment value (default is 10) with each subsequent line incremented by the current increment value.

Format 2 of the RENUMBER command will resequence all of the program in memory from the beginning of the program through the end of the program. The resulting program will have its first line numbered according to the <first> line number as specified with each subsequent line incremented by the default increment value.

Format 3 of the RENUMBER command will resequence all of the program in memory from the beginning of the program through the end of the program. The resulting program will have its first line numbered according to the <first> line number as specified with each subsequent line incremented by the <increment> as specified.

Format 4 of the RENUMBER command will resequence that portion of the program in memory as specified by the <start> parameter through the end of the program. The resulting program will have that <start> line numbered according to the <first> line number as specified with each subsequent line incremented by the current increment value.

Format 5 of the RENUMBER command will resequence that portion of the progarm in memory as specified by the <start> parameter through the line specified by the <end> parameter. The resulting program will have that <start> line numbered according to the <first> line number as specified with each subsequent line incremented by the <increment> as specified.

Formats 4 and 5 of the RENUMBER command will not allow you to resequence a program such that the result would cause lines to be merged. For example, a program with lines consecutively numbered from 10 through 100 could not be renumbered with RENUMBER 20 5 50 100 as this would cause lines 50 through 100 to collide with other existing lines. When this is attempted the error message "Renumber Range Error" is displayed.

All of the formats of the RENUMBER command will adjust all references in the program from the old line numbers to the new line numbers. This includes references made by the statements: ELSE, GOSUB, GOTO, IF, ON ERROR, ON GOTO, ON GOSUB, RESTORE, RESUME, RETURN, and THEN. Additionally, relational expressions with the function ERL on the left of the relation with a integer literal on the right (line number) will be adjusted.

Statements that previously referenced an undefined line number will be adjusted to reference an undefined line in the same relative location as before. For example, a program with lines consecutively numbered from 10 through 100 by 10s with a line reference to line 11 (non-existent) that is renumbered will have that line

reference adjusted to line 15.

Because good, complete examples of program renumbering would be quite lengthy none will be given. Instead, it is suggested that you "play" with the command on one of your own programs. Be sure to save the program on disk if it is a program that you do not want renumbered.

8.21 RUN Command

1 RUN

2 RUN <program name>

3 RUN <starting line>

4 RUN <program name> <starting line>

Where:

<program name> ::= <file name>[.<file type>][:<file disk>]
<file type> ::= BASICOBJ
<starting line> ::= <line number>

See also: LOAD command

Purpose:

The RUN command allows the user to execute a program already in memory or one stored on disk.

Comment:

When <program name> is not specified, the program currently in memory is executed, starting with the first line of the program, or at the line number specified.

Before the RUN command is executed, a CLEAR command is automatically executed.

<program name>, when specified, may be a string literal or an unquoted string literal. <program name> may not be a variable.

When the <program name> is specified, a search is made for the program. If the program is not found, the error message 'File Not Found' is displayed. If the program is found, a NEW command is executed and the specified program is loaded. Execution begins with the smallest line number, or at <starting line>, if specified.

<starting line> may be a line number that does not exist in the referenced program, in which case execution will begin at the first line greater than or equal to the specified line number.

Examples:	Explanation:
-RUN	Program in memory is executed.
-RUN TEST	Program "TEST" is loaded and executed.

8.22 SAVE Command

1 SAVE

2 SAVE <program name>

Where:

```
<program name> ::= [<file name>][.<file type>][:<file disk>]
<file type> ::= BASIC
                BASICOBJ
```

See also: LOAD and NAME commands

Purpose:

The SAVE command allows the user to save a program as a disk file.

Comment:

The entire <program name> operand is optional, and when omitted, the program will be saved under the name that it was LOADed, CHAINed, LINKed, or RUN under. If a name is not currently defined and the operand is omitted, an "Invalid Program Name" error will result.

<File disk> is optional--when not specified drive A will be used. The <file type> defaults to BASICOBJ unless the program already has a name with a file type of BASIC.

The program name, type, and disk will be displayed on the terminal after the program has been successfully written to disk.

When a file already exists with the same file description that file will be renamed to have a file type of BACKUP.

Examples:	Explanation:
-SAVE TEST:A "TEST.BASICOBJ:A" save	The program in memory will be written to disk and given the name 'TEST.BASICOBJ:A'.
-SAVE ..... SAVE	The program will be saved under the same name as loaded, i.e., the program will be updated on disk.
-SAVE TEST:S "TEST.BASICOBJ:S" saved	The program in memory will be written to disk and given the name 'TEST.BASICOBJ:S'

Incorrect Examples:	Explanation:
-SAVE	Program name must be specified if there is no prior LOAD, CHAIN, LINK, or RUN executed.
-SAVE TEST:T	Invalid unit specified.

8.23 STEP Command

1 STEP

2 STEP <count>

See also: BREAK, TRACE, UNBREAK, UNTRACE, and VARS commands

Purpose:

The STEP command allows the program to "single step" through the execution of the program.

Comment:

Format 1 of the STEP command causes the next statement in the program to be executed and a debugging break occurs.

Format 2 of the STEP command causes the next <count> statements in the program to be executed and a debugging break occurs.

Note that the STEP command operates on statements, not lines. Therefore it is possible to single step through each statement in a multi-statement line.

Example:

```
-LIST
  10 FOR I%=1 TO 3
  20       PRINT I%
  30       NEXT
-STEP
Break at line 20
-STEP
 1
Break at line 30
-STEP
Break at line 20
-STEP 3
 2
 3
Break at line 30
-
```

8.24 Top Command

TOP

Purpose:

The top command positions to the first line in the program and displays that line.

Comment:

The top command cannot be used if the program in memory is read protected (see LOAD command).

Examples:

```
Stop at line 0020
-<CR>
  20 MIDDLE% = LINE(0)/2.
-TOP
  10 REM Program: SAMPLE
-
```

8.25 TRACE and UNTRACE Commands

1 TRACE

2 TRACE VARS

3 UNTRACE

Purpose:

The TRACE and UNTRACE commands allow the programmer to trace the line numbers being executed by a program.

Comment:

Format 1 of the TRACE command turns the line number display on during execution.

Format 2 of the TRACE command turns the line number display on during execution and causes the display of all variables changed during the execution of each statement.

The UNTRACE command turns the line number display off during execution. This is the normal mode of program execution.

When a program is being traced each statement that is executed causes the line number of the statement to be displayed on the left hand sided of the console, in angle brackets. When TRACE VARS is in effect and a variable is changed by a statement the variable name and value that it was set to will be displayed on the left hand side of the console, in angle brackets.

Each statement of a multi-statement line, when executed, causes the line number to be displayed. The second and subsequent statements in a multi-statement line will be indicated by an offset count after the line number, indicating the relative offset of the start of that statement, from the start of the line. This offset value relates to the offset in the compressed, internal format, not the displayed format of the line. Nevertheless, this value is helpful in determining which statement of the multi-statement line is being executed.

Example:

```
0010 GOSUB 100 \ PRINT TOTAL
0020 FOR I% = 1 TO 3
0030       PRINT I%
0040       NEXT
0050 STOP
0100 TOTAL = 4.34 \ RETURN
-TRACE
-RUN
<10>
<100>
<100,18>
<10,5> 4.34
<20>
<30> 1
<40>
<30> 2
<40>
<30> 3
<40>
<50>
Stop at line 50
```

Explanation:

```
Execute line 10, GOSUB statement
      "      "  100, LET statement
      "      "  100, RETURN statement
      "      "  10, PRINT statement
      "      "  20, FOR statement
      "      "  30, PRINT statement, 1st time
      "      "  40, NEXT statement, 1st time
      "      "  30, 2nd time, prints 2
      "      "  40, 2nd time
      "      "  30, 3rd time, prints 3
      "      "  40, 3rd time
      "      "  50, STOP statement
```

```
-TRACE VARS
-RUN
<10>                                    Execute line 10, GOSUB statement
<100>                                      "      "  100, LET statement
  <TOTAL = 4.34>                        Variable changed
<100,18>                                Execute line 100, RETURN statement
<10,5> 4.34                                "      "   10, PRINT statement
<20>                                       "      "   20, FOR statement
  <I% = 1>                              Variable changed
<30> 1                                  Execute line 30, PRINT statement, 1st time
<40>                                       "      "   40, NEXT statement, 1st time
  <I% = 2>                              Variable changed
<30> 2                                  Execute line 30, 2nd time, prints 2
<40>                                       "      "   40, 2nd time
  <I% = 3>                              Variable changed
<30> 3                                  Execute line 30, 3rd time, prints 3
<40>                                       "      "   40, 3rd time
  <I% = 4>                              Variable changed
<50>                                    Execute line 50, STOP statement
Stop at line 50
-UNTRACE
-
```

8.26 UNBREAK Command

```
1 UNBREAK
2 UNBREAK AT <line reference>
3 UNBREAK ON <variable>

Where:

     <line reference> ::= <line number>
                          <line label>

See also: BREAK, STEP, TRACE, UNTRACE, and VARS commands
```

Purpose:

The UNBREAK command clears break points set by the BREAK command.

Comment:

Format 1 of the UNBREAK command will clear all break points currently set.

Format 2 will clear all break points referring to the specified line reference.

Format 3 will clear all break points currently set, referencing the specified variable.

For an example see the BREAK command.

8.27 Up Command

1 <up arrow>

2 <control/Z>

See also: Down, and LIST commands

Purpose:

The Up command allows you to backup and display the previous line in the program.

Comment:

The up command cannot be used if the program in memory is read protected (see LOAD command).

The actual key that you should enter to perform this function is dependant upon the currently set value for the UP key (refer the the OASIS System Reference Manual, "SET Command") and the console class code.

When the up command is entered the current line pointer is adjusted one line backward and the current line is displayed.

Attempting to executed the up command when the current line is at the first line of the program the message TOF: is displayed indicating that you are at the top of file.

8.28 VARS Command

1 VARS

2 VARS <variable list>

Where:

<variable list> ::= <variable name>[,<variable list>]

See also: BREAK, STEP, TRACE, UNBREAK, and UNTRACE commands

Purpose:

The VARS command allows the programmer to easily see the status of all variables defined in a program.

Comment:

Format 1 of the VARS comand causes each variable currently defined in the program to be displayed on the console, one variable per line, along with the contents of the variable. The sequence in which the variables are listed is the inverse sequence that the variables were initially defined in.

Format 2 of the VARS command causes each variable in the list to be displayed, one variable per line, along with the contents of the variable.

Dimensioned arrays are displayed one element per line.

Example:

```
-VARS
A$ = "ABCDEFG"
I% = 12
R1 = 12.34
R1$ = "TOTAL"
R3 = 1.234567
Y(1) = 1
Y(2) = 2
Y(3) = 3
Y(4) = 22

-VARS A$,R1$,R3
A$ = "ABCDEFG"
R1 = 12.34
R3 = 1.234567

-
```

8.29 XREF Command

1 XREF

See also: LPXREF command

Purpose:

The XREF command allows you to display all of the variables and lines used or referenced in the program.

Comment:

The XREF command cannot be used if the program in memory is read protected (see LOAD command).

The XREF command lists the program in memory on the console and then lists two tables of cross references for the program.

The first table lists all line numbers referenced and line labels defined or referenced along with the line number of the statement referencing the line number or label. In the table of references to line labels the line number of the line defining the label will have a colon following the line number.

The second table lists all variables and constants referenced in the program, in alphabetic order, followed by the line number of the statement with the reference to the variable or constant. A statement with multiple references to the same variable or label will have multiple occurrences of the line number in the table.

Array names are denoted by a pair of parentheses following the array name.

Each of the line number references in the second table will be followed by a single letter code indicating the type of reference to the variable or constant:

R Term used in an input type statement (INPUT, LINPUT, LINPUT USING, MAT INPUT, MAT READ, READ, READNEXT, and GET).

W Term used in an output type statement (DELETE, MAT PRINT, MAT WRITE, PRINT, PRINT USING, PUT, and WRITE).

M Term was modified by statement (LET, FOR, and MAT).

All other types of statements are unmarked.

The variables and constants are listed in the following sequence: variables, string constants, floating point constants, and integer constants.

Example:

```
-XREF

  10 OPEN #1: "NAME.DATA", INPUT SEQUENTIAL
  20 LOOP: PRINT CRT$("C");
  30       I% = 0
  40       INPUT: LINPUT #1: A$
  50            IF EOF(1) THEN GOTO EXIT
  60            PRINT AT$(6,I%+3);EXT$(A$,1,0);
  70            PRINT AT$(6,I%+6);EXT$(A$,4,0);
  80            I% = I%+5 \ IF I%+5<23 THEN GOTO INPUT
  90            WAIT
 100            GOTO LOOP
 110 EXIT: END

Line/Label          References

EXIT:                 50  0110:
INPUT:                40:   80
LOOP:                 20:  100
```

```
Variable/Constant  References

A$                 40R   60W   70W
I%                 30M   60W   70W   80M   80    80
"C"                20W
"NAME.DATA"        10
0                  30    60W   70W
1                  10    40R   50    60W
3                  60W
4                  70W
5                  80    80
6                  60W   70W   70W
23                 80
```

==========

CHAPTER 9

STATEMENTS

This chapter discusses each statement in a separate section. Each statement is described in four subsections:

1. **General form:** defines the syntax of the specific statement. For visibility this information is placed in a box at the top of the page. Note: the characters ::= should be read as "is defined as".

2. **Purpose:** one or two sentences that summarizes the purpose or general function of the statement.

3. **Comment:** detailed description of the statement specifying any restrictions, exceptions or errors that may occur.

4. **Examples:** general examples of the various forms of the statment if applicable.

For the convenience of novice programmers the BASIC statements are listed below by logical groups. In the body of this chapter, however, the statements are listed in alphabetic sequence, for quick reference purposes.

An appendix at the back of this manual lists all of the statements with their general syntax requirements.

A. Control and/or Branching Statements

CASE	Used with SELECT
CEND	Used with SELECT
ELSE	Used with IF
END	Exits program
FNEND	Marks end of user defined function
FOR	Loop control
GOSUB	Execute subroutine
GOTO	Unconditional branch
IF	Test expression-branch or execute depending on result
IFEND	Marks end of multi-line IF
NEXT	Used with FOR
ON ERROR	Invokes user written error handling routine
ON GOSUB	Selects subroutine depending upon value
ON GOTO	Selects branch depending upon value
OPTION	Set various options
OTHERWISE	Used with SELECT
QUIT	Exits BASIC
RESTORE	Resets DATA pointer
RESUME	Exits user written error handling routine
RETURN	Exits subroutine
SELECT	Specifies value that determines statements to be executed
SLEEP	Suspends processing for period of time
STOP	Exits program
THEN	Used with IF
WAIT	Pauses at bottom of screen display
WEND	Marks end of WHILE structure
WHILE	Executes statements while expression is true

B. Assignment and Declaration Statements

CLEAR	Erase variables from memory
COMMON	Defines variables used between program modules
DATA	Defines data constants
DEF	Defines user defined function
DIM	Allocates array space
LET	Assigns value to variable
MAT	Assign values to arrays

C. File Input and Output Statements

CLOSE	Closes file
DELETE	Erase record from file
GET	Get data from I/O devices
INPUT	Accepts ASCII data from file
LINPUT	Accepts line of ASCII data from file
LINPUT USING	Accepts line of ASCII data with control
MAT INPUT	Accepts ASCII data from file-assigns to array
MAT PRINT	Outputs ASCII data to file from array
MAT READ	Accepts data from file-assigns to array

MAT WRITE	Outputs data to file from array
MOUNT	Allows change of disk
OPEN	Opens file for subsequent input and output
POKE	Modifies memory
PRINT	Outputs ASCII data to file
PRINT USING	Outputs formatted ASCII data to file
PUT	Puts data to I/O devices
READ	Accepts data from file
READNEXT	Accepts data from indexed file
UNLOCK	Release record for other users use
WRITE	Outputs data to file

D. Program Linkage Statements

CHAIN	Branches to another program
CSI	Executes system program
LINK	Branches to another program
RUN	Branches to another program

E. Other Statements

empty or null statement
RANDOMIZE
REM

9.1 CASE Statement

1 CASE <expression>

See also: CEND, OTHERWISE and SELECT statements

Purpose:

The CASE statement is part of the SELECT-CASE-CEND programming structure that allows conditional execution of statements in a structured manner.

Comment:

The form and function of the CASE statement depends upon which format of the SELECT statement was used at the beginning of the SELECT-CASE-CEND structure. Format 1 of the SELECT statement requires that the CASE statements have relational expressions; format 2 of the SELECT statement requires that the CASE statements have expressions the match in type to the expression used in the SELECT statement--numeric with numeric, string with string.

When the CASE statement is used with format 1 of the SELECT statement the relational expression of the CASE statement is evaluated and, if true, the statements following the CASE statement will be executed.

When the CASE statement is used with format 2 of the SELECT statement the expression of the CASE statement is compared to the expression of the SELECT statement and, if true, the statements following the CASE statement will be executed.

When the evaluation of the CASE statement causes the statements following the CASE statement to be executed, execution will continue until another CASE, CEND, or OTHERWISE statement is encountered at the same level.

When the comparison is false the statements following are skipped until another CASE, CEND, or OTHERWISE statement is encountered for this SELECT-CASE-CEND structure.

SELECT-CASE-CEND structures may be nested to any level.

It is best to use different levels of indentation to illustrate the structure of a nested SELECT structure--the CASE statement does not indicate which SELECT expression is being used--only the BASIC execution module "knows" unless you use some form of documentation.

This programming structure should be used to replace complex IF-THEN-ELSE statements, ON-GOTO, and ON-GOSUB statements to produce a more structured program. It is much more versatile than the ON statement because the conditional execution is determined by a general expression rather than an integer expression with only positive, sequential values.

This structure is particularly useful for a menu tree when the controlling expression is a string.

Note: Any statements between a SELECT statement and the first CASE statement will never be executed unless they are branched to.

Note: The program should never branch out of a SELECT-CASE-CEND structure without executing the CEND statement as the internal SELECT stack will not be cleaned up which will result in un-necessary memory usage.

Example:

```
0010 INPUT CONTROL$
0020 SELECT CONTROL$
0030     CASE ""
0040          PRINT CONTROL$
0050          GOSUB 1000
0060     CASE "HELP"
0070          GOSUB 2000
0080          QUIT
0085     CASE CONTROL$   If control = control (always true)
0087          PRINT "Invalid input";
0090     CEND
0100 SELECT
0110     CASE CONTROL$=""
0120          PRINT CONTROL$
0130          GOSUB 1000
0140     CASE CONTROL$="HELP"
0150          GOSUB 2000
0160          QUIT
0170     OTHERWISE
0180          PRINT "Invalid input"
0190     CEND
```

Explanation:

```
Accept control value
Using this control value then:
If control is null execute following
  else skip to line 60
Only executed when CONTROL$ is empty
 "     "      "      "
If control is "HELP" execute following
  else skip to next CASE or CEND
Only executed when CONTROL$="HELP"
 "      "      "      "
End of select structure
Same as above
```

9.2 CEND Statement

1 CEND

See also: CASE, OTHERWISE and SELECT statements

Purpose:

The CEND statement is part of the SELECT-CASE-CEND programming structure that allows conditional execution of statements in a structured manner.

Comment:

The CEND statement marks the end of the SELECT structure.

There is only one CEND for each SELECT.

Example:	Explanation:
1010 SELECT OPTION$	Using variable OPTION$
1020 CASE "HELP"	
1030 GOSUB DISPLAY.HELP	Perform if OPTION$="HELP"
1040 CASE "INIT"	
1050 GOSUB INIT.VAR	Perform if OPTION$="INIT"
1060 GOSUB INIT.FILE	" " "
1070 CASE "PRINT"	
1080 DEVICE.NUM% = 16	Perform if OPTION$="PRINT"
1090 CASE "TYPE"	
1100 DEVICE.NUM% = 15	Perform if OPTION$="TYPE"
1110 CEND	
1120 RETURN	Perform always

9.3 CHAIN Statement

1 CHAIN <program name expr>

Where:

<program name expr> ::= <file name>[.<file type>][:<file disk>]
<file type> ::= BASICOBJ (with BASIC)
BASICCOM (with RUN)

See also: CLEAR, LINK and RUN statements

Purpose:

The primary use of the CHAIN statement is to link together BASIC program segments.

Comment:

The CHAIN statement terminates the execution of the program in which it is encountered, loads the program indicated, and continues execution at the beginning of the program segment.

The CHAIN statement will close all open channels (files) and all variables that have not been defined as COMMON variables will be cleared from memory.

Previous versions of OASIS BASIC supported a <line number> operand. The recommended method of transferring control to another program at a specific line is the use of a control variable (defined as COMMON) that is tested by an ON-GOTO statement at the start of the program transferred to.

The <program name> must be a valid string expression. If the program cannot be found in the directory, a non-trapable error will occur.

Note: When the RUN version of BASIC is being used (execution of compiled programs only) only programs that have been compiled and have a file type of BASICCOM will be searched for by this command.

The CHAIN statement will "wrap up" all active programming structures: FOR-NEXT, WHILE-WEND, IF-THEN-ELSE, IF-IFEND, SELECT-CASE-CEND, and the ON ERROR will be turned off.

The CHAIN, RUN, and LINK statements all perform similar tasks, but with significant differences:

Program Linkage Statements

Statement	I/O Channels	Variables	COMMON
RUN	Closed	Cleared	Cleared
CHAIN	Closed	Cleared	Not cleared
LINK	Not closed	Cleared	Not cleared

Examples:	Explanation:
0010 CHAIN "SEGM01"	Program named 'SEGM01' will be loaded, all files will be closed, & control will pass to the first statement of 'SEGM01'.
0030 CHAIN "SEGM0"&NUM(I)&":S"	When I equals 1, this statement is the same as example 10. If I is equal to 3, program 'SEGM03' will be executed, etc.

9.4 CLEAR Statement

1 CLEAR

2 CLEAR <variable list>

Where:

```
<variable list> ::= <numeric variable>[,<variable list>]
                    <string variable>[,<variable list>]
                    <array name>[,<variable list>]
```

See also: CHAIN, COMMON, LINK and RUN statements

Purpose:

The CLEAR statement initializes the working storage area.

Comment:

The CLEAR statement effectively erases all variable names and their contents from memory.

Variables defined as COMMON variables are not erased by this command.

This operation is performed automatically whenever a CHAIN, LINK, LOAD, NEW, or RUN command is executed. It may be necessary to use this separate command when there are many variables defined in working storage that are not going to be used again, and there are no variables whose loss would be detrimental to program execution.

The main advantage gained is a fresh work area that may allow the program to continue execution that, without it, might have required more memory than available.

Optionally this statement may clear specific variables (or complete arrays) from memory when they are no longer needed by the program.

Examples:	Explanation:
0010 CLEAR	All variables are cleared from memory.
0020 CLEAR A,B,INDEX%	Only the variables A, B, and INDEX% are cleared from memory (unless they were defined as COMMON).
0030 CLEAR ARRAY1$,A,B	The entire array ARRAY1$ is erased from memory along with the variables A and B.

9.5 CLOSE Statement

1 CLOSE #<channel>

Where:

<channel> ::= <integer expression>

See also: CHAIN, CSI, END, MOUNT, OPEN, QUIT and RUN statements

Purpose:

The CLOSE statement is used to terminate I/O between the BASIC program and a data file.

Comment:

The CLOSE statement causes the output of the last block of data to the file. Execution of a CHAIN, END, or CSI statement automatically closes all open files. The RUN command automatically closes any open files before execution begins. The QUIT command will close any open files before exiting BASIC.

The <channel> must have the same value as that used with the OPEN statement.

Once a file has been closed, it may be reopened on any available channel number.

If the user should happen to abort a BASIC program (by an IPL or power failure) when indexed or sequential files are open, errors may exist in the file directory or in the file itself. It is acceptable to abort the program by using the Program Cancel-key or the System Cancel-key, but exiting by a system reset button or power failure can be disasterous.

Examples:	Explanation:
0010 CLOSE #1	File opened on channel one is closed.
0020 CLOSE #INPUT%	File opened on channel corresponding to value of variable INPUT% is closed.

Incorrect examples:	Explanation:
0010 CLOSE "IVY MASTER A"	File names are not allowed.
0020 CLOSE #4,#5,#6,#10	Multiple channels not allowed.
0030 CLOSE #17	Channel 17 is invalid.

9.6 COMMON Statement

1 COMMON <variable list>

Where:

```
<variable list> ::= <simple variable>[,<variable list>]
                    <dim variable>[,<variable list>]
<dim variable> ::= <array name>(<dimension>[,<dimension>])
<dimension> ::= <numeric expression>
```

See also: CLEAR, DIM, OPTION, and RUN statements

Purpose:

The COMMON statement allows you to specify that certain variables are shared between segments of a program and are, therefore, not to be cleared.

Comment:

The COMMON statement must be the first statement on a line--there can be no line label specified on the same line as a COMMON statement.

The COMMON statement is an executable statement, similar to the DIM statement--in fact, it must be executed before any references are made to the variables it is defining as common.

When a program is RUN, CHAINed to, or LINKed to, the entire program is scanned for any and all COMMON statements. When one is found the variables specified on that statement are searched for in the COMMON variable storage. When a variable is found it will be left as is. When a variable is not found in the COMMON variable storage area it will be defined or dimensioned in that area.

Note: If a variable was used in a previous program but not defined as COMMON before its use, the value will not be retained at the time it is defined as COMMON.

Although it is not necessary to re-define all of the variables that are COMMON between programs it is definitely a good programming practice. It is also not necessary to specify the variables in a COMMON statement in the same sequence as they might have been defined in a previous program's COMMON statement--variables are accessed by name, not location or sequence.

Examples:	Explanation:
0010 COMMON A,B,A%	The variables A, B, and A% were defined, or will be used, by another program.
0020 COMMON ARRAY$(5,22),CONTROL	Similar to above but also dimensions ARRAY$

9.7 CSI Statement

1 CSI <string-exp>

Purpose:

The CSI statement allows the BASIC program to execute any OASIS command, resuming execution of the BASIC program afterwards.

Comment:

All I/O channels will be closed when the CSI statement is executed.

<String-exp> is any valid OASIS command, with all arguments and options required by the specific command. Refer to the OASIS System Reference Manual for complete specifications of these commands.

When the first character of <string-exp> is the ">" character, the string will be displayed on the console device, otherwise the string will not be displayed (command executed in "silent" mode).

When the CSI statement is executed your BASIC program and all of its work areas are marked as protected memory. A special call to the operating system passes the string expression to the Command String Interpreter which executes the desired program. Upon completion of that program the operating system reloads the BASIC or RUN command, if necessary, unprotects the memory area containing your program, and continues execution of your program.

The CSI statement should not be used to execute large OASIS commands because of the restricted memory available. Additionally, when the following commands are executed the result will be unpredictable: DEBUG, ASSEMBLE, BASIC, and RUN. The ATTACH program cannot be called to attach a new device driver; however, it may be called to change some options on a currently attached device. In addition the stack of the EXEC language must be empty.

Examples:	Explanation:
0010 CSI "LIST CUSTOMER MASTER (PRINT NOHEAD)"	The file CUSTOMER MASTER is printed without headings.
0015 A$=">filelist a (exec append)" 0020 CSI A$	Command is displayed. The file named SELECTED.EXEC is appended with the current filelist from the directory of the A disk.

9.8 DATA Statement

1 DATA <data list>

Where:

```
<data list> ::= <data element>[,<data list>]
<data element> ::= <numeric constant>
                   <quoted string constant>
                   <unquoted string constant>
```

See also: READ and RESTORE statements

Purpose:

The DATA statement is used to define information to be read by the READ statement. The DATA and READ statements are useful for defining the initial contents of an array, etc.

Comment:

The DATA statement must be the first, and only statement on a line--there can be no line label specified on the same line.

The data elements in one or more DATA statements are used sequentially, in the order that they appear in the line, in the order that the lines appear in the program. (It is possible to re-use data elements--see RESTORE statement.)

When a data element is to contain leading or trailing spaces or embedded commas it must be defined as a quoted string constant.

This statement, along with the READ and MAT READ statement, is very useful for defining the initial values to be used for variables and array elements. It is much faster to perform a READ or MAT READ than it is to use the LET statement.

Examples:

```
10 DATA 1.23,2.34,3.45,LITERAL,ANOTHER LITERAL
20 DATA "He said, ""Bring the glass.""","T. J. Collins, Jr."
30 DATA 1,1,2,1,1,1,0,1,1,5,1,16,1,-1
```

9.9 DEF Statement

1 DEF FN<variable>(<arguments>) = <expression>

2 DEF FN<variable> = <expression>

3 DEF FN<variable>(<arguments>)

4 DEF FN<variable>

Where:

<variable> ::= <simple variable name>
<arguments> ::= <simple variable name>[,<arguments>]

See also: FNEND and LET statements

Purpose:

The DEF statement allows the programmer to define a user defined function.

Comment:

In some programs you may want to execute the same sequence of statements in several places. You can define a sequence of operations as a user-defined function and use this function like you use the functions BASIC provides.

The DEF statement has two basic forms: single line (formats 1 and 2); multi-line (formats 3 and 4).

The DEF statement must be the first statement on a line--there can be no line label specified on the same line as the DEF statement.

The <variable> following the characters 'FN' is independent from the program. The function is referenced by the complete name, including the FN characters.

Any variable referenced in the expression which is not an argument of that function has its current value in the user program.

The expression may include any valid element. It should be noted that a single line function should not reference itself as this causes an infinite loop.

The <argument> is a dummy argument: it has no relation to the program and cannot be changed by the program. If the dummy argument is also a variable used by the program, they are independent of each other.

The argument must be a simple variable name, that is, array referrences are invalid.

In the single line format of the DEF statement, the variable and the expression must match in type, i.e., string variable with string expression, or numeric variable with numeric expression.

During execution the expression is analyzed and the value is assigned to the function. This value takes the place of the function call in the expression that references the function.

In the multi-line forms of the DEF statement there must be a corresponding FNEND statement to mark the end of the function definition.

In the multi-line forms of the DEF statement the statement following the DEF statement are executed until the FNEND statement is encountered, at which time the value of the function is returned and execution resumes at the location of the function reference. The value of the function is assigned by a LET statement in the function definition: LET FN<variable> = <expression>. There can be more than one of these assignment statements in a function definition but only the last one executed will be the assignment used.

There can be only one FNEND statement for each multi-line function definition.

Most statements can be used within the function definition (between the DEF and FNEND statements). However, transfers into or out of the definition (with GOTO or

GOSUB) should not be used. (There are no restrictions in this regards except that the FNEND statement cannot be executed without performing a multi-line function reference.)

DATA statements may be READ from a statement within a function definition.

A multi-line function definition that does not execute an assignment statement assigning the value of the function will return the last defined value of the function.

The DEF statement may be placed anywhere in the program, however, it is executed only when referenced by another statement.

You may not re-define a DEF function. No error occurs when this is attempted but only the first definition is used.

==========

Examples:

```
0010 DEF FNA(X) = SQR(X^2+Y^2))-SQR(X)
0020 B = 2
0030 C = 5
0040 Y = 10
0050 Z = FNA(B)/FNA(C)
0100 Z = (SQR(B^2+Y^2)-SQR(B))/(SQR(C^2+Y^2)-SQR(C))
1000 DEF FNX%(A,B)
1010      IF A>B THEN FNX%=A*B+3.4  GOTO 1030
1020      FNX% = A*B
1030      FNEND
1040 PRINT FNX%(3.1,2.3)
```

Explanation:

X is the dummy argument. The value of Y is taken from the program, 10 in this example. Each of the two calls to this function in line 50 cause the value of the argument, 2 in the first call and 5 in the second call, to take the place of any and all references to that dummy variable in the expression of the function. The function is evaluated. Line 100 is an equivalent statement as line without using the function calls. As can be seen this is not only more difficult to read, but when there are more references to the same function there will be more code involved.

Multi-line def, arguments of A, B
Defines value of function and exits.
Define value of function
End of definition
Will print 10.

Incorrect examples:

```
0010 DEF FNA(S^2) = 2*S+S
0020 DEF FNA$(B) = 2*B
```

Explanation:

Dummy argument must be a simple variable.
Function name must match expression in type (string or numeric).

==========

9.10 DELETE Statement

1 DELETE #<file>,<key>

Where:

<file> ::= <integer expression>
<key> ::= <string expression>
<numeric expression>

Purpose:

The DELETE statement deletes a specified indexed or direct record from an open file.

Comment:

<file> is the channel number of an open, indexed or direct, disk file, with access mode of OUTPUT or UPDATE, and an access method of DIRECT or INDEXED.

<key> is a string expression representing the key of the indexed record to be deleted or a numeric expression representing the record number of the direct record to be deleted. A string key is required if the I/O channel was opened with access method INDEXED, and a numeric key is required if the I/O channel was opened with access method DIRECT.

The record specified by the <key> is removed from the file and the EOF indicator is set off.

When the record is not found the sequential access pointer and the EOF indicator will be the same as if the record were found and deleted.

Examples:	Explanation:
0010 OPEN #1: F$,UPDATE INDEXED	
0020 OPEN #2: F1$,UPDATE DIRECT	
0030 DELETE #1,"0001"	Record with key "0001" is deleted.
0040 DELETE #2,30	Record number 30 is deleted.

Incorrect Examples:	Explanation:
0020 DELETE #1,25	Indexed files use string keys.
0030 DELETE #1,K$:A1$,A2$	Record variables not allowed.

9.11 DIM Statement

1 DIM <dim variable list>

Where:

```
<dim variable list> ::= <dim variable>[,<dim variable list>]
<dim variable> ::= <simple variable>(<num expr>[,<num expr>])
```

See also: COMMON, MAT, MAT INPUT, MAT PRINT, MAT READ, MAT WRITE, and OPTION statements

Purpose:

The DIM statement instructs the system to reserve storage space for an array by specifying a maximum subscript (dimension).

Comment:

The DIM statement is an executable statement. In fact, it has to be executed in order to be effective.

Numeric or string variables may be dimensioned with one or two dimensions. The maximum value for each dimension is 32767, however, the restraints of memory size usually limit this to a much lower value. An array may not be re-dimensioned.

When a variable is dimensioned a reference to the same variable name will refer to the array. This is only allowed with certain types of statements (i.e., MAT). In other statements the error "Inconsistent usage" will occur.

Any reference to an array beyond the allocated size will cause a subscript error.

Arrays are created with a zero element in each dimension, unless OPTION BASE 1 is in effect. For instance, if the array X were dimensioned X(5), there would be six elements in the array with subscripts of 0, 1, 2, 3, 4, and 5.

Examples:	Explanation:
0010 DIM X(20),Y(2,5),A$(5,5)	Array X has 21 elements, array Y has 18 elements, string array A$ has 36 elements.

Incorrect Examples:	Explanation:
0010 DIM X(2,2,2)	Can have only 2 dimensions.
0020 DIM Y(99999)	Maximum dimension is 32767.

9.12 ELSE Statement

1 ELSE [<statement>]

See also: IF, IFEND and THEN statements

Purpose:

The ELSE statement specifies the action to be taken when a multiline IF statement relation is not true.

Comment:

The ELSE statement is only valid as part of a multi-line IF statement, however, the ELSE verb may be used in a single line IF statement.

<statement> may be any valid statement or statements, including another IF statement. It should not, however, be an IFEND statement.

Examples:

```
0010 IF A
0020     THEN GOSUB 2000
0030          PRINT USING "###",A
0040          GOTO TOP.OF.PAGE
0050     IFEND
```

Explanation:

```
Test A for non zero
Perform if A<>0
   "    "   "
   "    "   "
End of conditional execution
```

```
0010 IF VALUE > CONTROL
0020    THEN IF VALUE > LIMIT
0030       THEN GOSUB ERROR
0040            GOTO EXIT
0050       ELSE IF ERR.NUM < ERR.LIMIT THEN QUIT
0060       IFEND
0070    IFEND
```

Explanation:

```
Test expression
Perform if expr is true
Perform if both expr are true
  "     "   "  "   "    "
Perform only if first expr is true
and second expr is false
End conditional execution from second expr
End of conditional execution
```

Incorrect Example:

```
0010 IF CONTROL<LIMIT THEN WAIT
0020      ELSE PRINT "ERROR"
```

Explanation:

Not in multi-line IF statement

9.13 END Statement

1 END

See also: STOP and QUIT statements

Purpose:

The END statement terminates execution of the program.

Comment:

The END statement, unlike the STOP statement, not only terminates execution of the program but also closes all open I/O channels and, if the RUN command is being used (not BASIC), exits from the BASIC environment; otherwise control returns to the command mode.

The END statement should be the last statement in a program. Although this is not required by OASIS BASIC it is required by ANSI and is a good programming practice because it can serve as an indicator that you intended it to be the end of the program.

You cannot CONTINUE after an END statement has been executed.

9.14 FNEND Statement

1 FNEND

See also: DEF statement

Purpose:

The FNEND statement marks the end of a multi-statement user defined function.

Comment:

The FNEND statement may only be used, and must be used, in a multi-line user definied function.

There may only be one FNEND statement for each multi-line function.

The statement between and including the DEF and FNEND statements are not executed unless referenced from a statement in the body of the program.

Example:

```
0010 DEF FNTEST(A)
0020      FNTEST=PI*A*A
0030      FNEND

0040 DEF FNCENTER$(STRING$,LENGTH)
0050      IF LENGTH=0 THEN FNCENTER$=""  GOTO 110
0060      IF LEN(STRING$)=0 THEN FNCENTER$ = SPACE$(LENGTH)  GOTO 110
0070      FILL = LENGTH-LEN(STRING$)
0080      IF MOD(FILL,2)=0
0090           THEN FNCENTER$ = SPACE$(FILL/2)&STRING$&SPACE$(FILL/2)
0100           ELSE FNCENTER$ = SPACE$(FILL/2)&STRING$&SPACE$(FILL/2+1)
0110           IFEND
0120      FNEND
```

Explanation:

Start of function definition

End of function definition

Start of function definition

End of function definition

9.15 FOR Statement

1 FOR <num index>=<start> TO <limit>

2 FOR <num index>=<start> TO <limit> STEP <increment>

3 FOR <index>=<expression list>

Where:

```
<num index> ::= <simple numeric variable>
<start> ::= <numeric expression>
<limit> ::= <numeric expression>
<increment> ::= <numeric expression>
<index> ::= <simple variable>
<expression list> ::= <num expr list>
                      <string expr list>
<num expr list> ::= <numeric expr>[,<num expr list>]
<string expr list> ::= <string expr>[,<string expr list>]
```

See also: NEXT statement

Purpose:

The FOR statement define a program loop and execute that loop until a terminating condition is met.

Comment:

The FOR statement assigns an initial value, <start>, to the index and saves the limiting value <limit>.

The STEP increment (format 2), is saved for use by the corresponding NEXT statement. If the STEP value is not specified (format 1), a value of +1 is used.

The following paragraphs pertain to formats 1 and 2 of the FOR statement:

Upon initial execution of the FOR statement, the index variable is assigned its initial value. The index variable is then compared to the limiting value and, if the index has not surpassed the limit, execution is passed to the statement following the FOR statement. When the index has surpassed the limit, execution is passed to the statement following the matching NEXT statement. If there is no matching NEXT statement an error occurs: "FOR without NEXT".

The STEP value, when specified, may be a negative value. When the STEP is positive, the limiting value must be greater than or equal to the initial value. When the STEP is negative the limiting value must be less than or equal to the initial value.

The value of the index variable surpasses the limiting value when it is more positive (for a positive STEP value) or more negative (for a negative STEP value).

A FOR NEXT loop is defined by the FOR and NEXT statements, with each statement marking the beginning and end of the loop.

FOR NEXT loops may be nested to any depth.

If a FOR NEXT loop is nested, it must be completely contained within the next higher FOR NEXT loop. An error will occur if the system detects an illegal form of nesting. A common practice to determine if your FOR NEXT loops are legal is to draw lines between the matching FOR and NEXT statements (see examples). If a line crosses another then it is an illegal form of nesting.

The FOR NEXT loop may be exited with a GOTO statement. When this is done, the FOR NEXT loop will remain open until another FOR NEXT loop is executed using the same index variable or when this loop is re-entered.

Upon termination of a FOR NEXT loop the index variable will retain the first value that exceeded the limiting value. For instance, the first example below will have the value +11 upon termination.

Format 3 of the FOR statement allows the loop to operate on a "set" of values with

the set being defined by the expression list. In this form the expressions must match in type (numeric or string) with the index variable.

In this format the index variable is initialized to the value of the first expression. There is no limit testing as there is no limit defined. Rather, the FOR NEXT loop is performed until the list of expressions is exhausted, with each execution of the matching NEXT statement causing the next expression to be evaluated and assigned to the index variable.

==========

Examples:

```
--- 10 FOR I=1 TO 10
|    .
|    .
|    .
--- 50 NEXT I
```

Explanation: Loop will execute 10 times.

```
--- 10 FOR I%=C+3 TO R*2 STEP .2
|    .
|    .
|    .
--- 50 NEXT I%
    60 . . .
```

Explanation: Initial value is 3 plus the current value of C. Limiting value is current value of R times 2. If limit is less than initial value the loop is not excuted and control will pass to line 60. If variables C or R are changed within the loop, initial value and limiting value will not be affected as they are evaluated only once.

```
----- 10 FOR I% . . .
|  -- 20 FOR J% . . .
|  |    .
|  |    .
|  --  50 NEXT J%
|  --  60 FOR J% . . .
|  |    .
|  |    .
|  --  100 NEXT J%
|      .
----- 150 NEXT I%
```

Explanation: This illustrates a correct form of nesting.

```
----- 10 FOR INDEX$="A","B","ABCD"&SPACE$(2)
|      .
|      .
----- 40       NEXT INDEX$
```

Explanation: Loop will execute 3 times. Variable must match FOR index variable, if used.

```
----- 10 FOR I%= 1,2,3,6,8,22,99
|  -- 20       FOR A$="A","B",X$
|  |
|  -- 40            NEXT A$
----- 50       NEXT I%
```

Explanation: Will execute 7 times. Will execute 3 times for each of the seven major loops. Terminate current loop. Terminate current loop (major).

Incorrect examples:

```
      10 FOR 1 TO 50 STEP 2
      20 FOR X = 1 STEP -3
      30 FOR J = 5 TO 1 STEP 2
      40 FOR K = 1 TO 4 STEP -5
```

Explanation: Index variable missing. Limiting expression is missing. Loop will fail initial test. Loop will fail initial test.

```
------ 10 FOR I . . .
|   --- 20 FOR J . . .
|   |    .
|   |    .
---+--- 50 NEXT I
   |    .
   --- 90 NEXT J
```

Explanation: Illegal nesting. Note that lines cross here.

==========

9.16 GET Statement

1 GET DEVICE <device number>,<variable list>

2 GET MEMORY <address>,<variable list>

3 GET PORT <port>,<variable list>

Where:

```
<device number> ::= <numeric expression>
<address> ::= <numeric expression>
<port> ::= <numeric expression>
<variable list> ::= <numeric variable>[,<variable list>]
                    <string variable>[,<variable list>]
```

See also: PUT and WAIT statements

Purpose:

The GET statement allows you to accept a single byte or list of bytes from an I/O device such as an analog to digital converter. The GET statement is also useful for accepting keyboard input, if available, and without any prompting or waiting for the operator.

Comment:

This statement is not available when using the RUN2 version of BASIC.

<device number>, <address>, or <port> is a numeric expression which is rounded up and integerized. <Device number> must be in the range of 9 through 32. This number is the address of a logical device (CONIN, CONOUT, PRINTER1, etc.). <Port> must be in the range: 0-255. This number is the address of the I/O port.

<address> must be in the range: -32767 - +32767. This value, unlike other integers, is evaluated as an unsigned integer which adjusts its range to 0 - 65565. It is best to use hexadecimal values for <address> as they are easily interpreted as unsigned integer values.

The data accepted from the port, device, or memory is mapped in a one to one relation with the variable list. If the variable is numeric it receives an eight bit integer. If the variable is a string, only one character is assigned to it. When more than one variable is specified each variable is evaluated independently of the others. When GET MEMORY is used with multiple variables the memory address is incremented by 1 for each byte accepted.

BASIC does not test to see if the I/O device is ready before accepting the input. When the device is not ready the data "accepted" will be null or zero.

The GET statement along with the PUT and WAIT statements discussed in their respective sections, provides a means of communicating with any device in the system. These statements would normally be used to access devices that are not supported by the operating system although there is no restriction in this regards. In fact, these statements may be used to destroy the system, so please...don't.

The GET DEVICE statement accepts a byte or bytes of data from the logical device driver specified. A table of the logical device driver numbers is included in the OASIS System Reference Manual. If your system is not interrupt driven you should not use this statement. If the device has no information ready a null or zero byte is returned.

The GET MEMORY statement reads the random access memory in the system. This statement could be used to read data stored by your own user written device driver. Because of the interpretation of the <address> as an unsigned value it is easiest to use hexadecimal values (see section on "Integer Constants" at the beginning of this manual and the section on "Numeric Functions").

The GET PORT statement accepts a byte or bytes of information from a physical port. All devices have port numbers, usually determined by the hardware interface electronics. If you have a reason to use this statement you would already know the port number of the device that you wished to access. If the port has no information ready a null or zero value is returned.

The GET DEVICE statement is useful on system with an interrupt driven console. Sometimes you need to accept a reply from the operator that he wasn't expecting to be asked (error message response). In this situation it would be desirable to make sure that the "type ahead buffer" was cleared before asking for the operator response. See example line 50 for a method of doing this.

Examples:	Explanation:
0010 GET MEMORY 0800H,A$,B$	Two bytes of data from memory address 0800 and 0801 hex are assigned to A$ and B$, respectively.
0020 GET DEVICE 32,A,B,C,D	Four bytes of data from device #32 are assigned to the numeric variables A,B,C, and D respectively.
0030 GET DEV 9,A$	Gets one character from the console and assignes it to A$.
0050 GET DEV 9,A%IF A% THEN 50	This line will get information stored in the console input buffer until that buffer is empty (null returned).

9.17 GOSUB Statement

1 GOSUB <line reference>

2 GO SUB <line reference>

Where:

<line reference> ::= <line number>
<line label>

See also: ON and RETURN statements

Purpose:

The GOSUB statement transfers control to the specified line.

Comment:

The GOSUB statement eliminates the need to repeat frequently used groups of code in a program. Such a group of statements is a subroutine. The subroutine must logically end with a RETURN statement.

The subroutine may contain GOSUB statements, even a GOSUB to itself. This is called a recursive subroutine. There is no limit on the number of unreturned subroutines in progress, however, each unreturned subroutine requires 5 bytes of memory.

When a GOSUB statement (or an ON-GOSUB statement) is executed, BASIC saves the location of the statement that physically follows. Upon execution of the RETURN statement, control transfers to the statement whose location was saved.

Example:	Explanation:
0010 GOSUB 100 \ A = A+1 . . 0100 PRINT A . . . 0150 RETURN	Subroutine starting at line 100 will be executed. Upon return from the subroutine th the variable A will be incremented. Control will be transfered to the statement following the 'GOSUB' that called this subroutine.
1000 GOSUB INPUT	The subroutine starting with the label INPUT will be executed with execution resuming with the line following 1000 when the subroutine's RETURN statement is executed.

Incorrect example:	Explanation:
0010 GOSUB 100 \ A = A+1 0020 LET X = SQR(Y(4/7)) . . . 0100 PRINT A 0110 GOTO 20	Should not exit from a subroutine except with a 'RETURN' statement.

9.18 GOTO Statement

1 GOTO <line reference>

2 GO TO <line reference>

Where:

<line reference> ::= <line number>
<line label>

See also ON and ON ERROR statements

Purpose:

The GOTO statement transfers control, unconditionally, to a specified line.

Comment:

GOTO must be followed by a line reference of a line that exists. This line reference may be of a line of a non-executable statement. If so, control will pass to the first executable statement following the referenced line. When the line referenced does not exist, an error will occur.

A GOTO statement should not be used to jump into the middle of a FOR-NEXT loop because a "NEXT without FOR" error will occur. If it is necessary to branch into a FOR-NEXT loop to save coding, a switch should be used that will bypass the NEXT statement.

Similarly, a GOTO should not be used to jump into the middle of a subroutine, WHILE-WEND and SELECT-CASE-CEND structures.

The GOTO statement can be entered as the two words GO TO.

Examples:	Explanation:
0010 GOTO 1020	Control is unconditionally transferred to line 1020.
0020 GOTO BEGIN	Control is unconditionallly transferred to the line with the label BEGIN.

Incorrect examples:	Explanation:
0020 GOTO 20	Infinite loop - this is not detectable by BASIC.
0030 IF I > 4 THEN I = I-1 \ PRINT I \ GOTO 30	Valid single line loop.

9.19 IF Statement

1 IF <expression> THEN <then clause> ELSE <else clause>

2 IF <expression> THEN <then clause>

3 IF <expression>

Where:

```
<expression> ::= <arithmetic expression>
                 <logical expression>
                 <relational expression>
<then clause> ::= <statement>
                  <line number>
                  <empty statement>
<else clause> ::= <statement>
                  <line number>
<line reference> ::= <line number>
                     <line label>
```

See also: ELSE and THEN statements

Purpose:

The IF statement provides for the conditional execution of a statement or statements or the conditional branching to a different section of code.

Comment:

<statement> may be any valid BASIC statement.

In the IF statement, formats 1 and 2, the <expression> is first tested. If the result is non-zero, then the THEN clause receives control. Since <statement> may be multiple statements separated by backslants, control will be retained by these statements until the end of line or a matching ELSE is encountered. When this occurs, control will pass to the line following the IF statement.

If the result is zero, a search is made for a matching ELSE; when found, control will pass to the statement or line number following the ELSE term. When no matching ELSE is found, control will pass to the line following the IF statement.

Any ELSE term encountered by BASIC is assumed to match to the most previous, unmatched THEN clause. Tabs and indentation are not considered by BASIC in determining matching THEN ELSE clauses, they are only for use as a programming aid in the intended structure of the code.

Format 3 of the IF statement provides for complex, multi-line IF statements, where the other lines contain THEN and/or ELSE statements. This multi-line structure is terminated by the IFEND statement.

In any format, the IF statement may be nested up to 127 levels.

Examples:	Explanation:
`0010 IF A=1 THEN PRINT "A = 1"`	When variable A is equal to 1, the literal 'A = 1' is printed; otherwise control passes to the line following.
`0020 IF A=1 THEN PRINT 'OK' ELSE 30`	When variable A is equal to 1, the literal 'OK' is printed; otherwise line 30 is executed.
`0030 IF INP THEN GOSUB 1000`	When the value of the function INP is greater than zero, the subroutine at line 1000 is executed; otherwise control is passed to the line following.

```
0050 IF VALUE>0 THEN PRINT "POSITIVE" ELSE PRINT "NEGATIVE"
                                        When the variable VALUE is positive,
                                        the literal 'POSITIVE' will be
                                        printed; otherwise the literal
                                        'NEGATIVE' is printed.
0060 IF "A"="A " THEN 70                Unequal length strings being
                                        compared. Will test false.
1010 IF A                               Test A for non zero
1020     THEN GOSUB 2000                Perform if A<>0
1030          PRINT USING "###",A       "    "   "
1040          GOTO TOP.OF.PAGE          "    "   "
1050     IFEND                          End of conditional execution

1210 IF VALUE > CONTROL                 Test expression
1220      THEN IF VALUE > LIMIT         Perform if expr is true
1230                THEN GOSUB ERROR    Perform if both expr are true
1240                     GOTO EXIT      "    "   "    "    "    "
1250                ELSE IF ERR.NUM < ERR.LIMIT THEN QUIT Single line IF
                                        Perform only if first expr is true
                                        and second expr is false
1260                IFEND               End conditional execution from second expr
1270      IFEND                         End of conditional execution
```

```
Incorrect examples:                     Explanation:

0010 IF D THEN X = A+D GOTO 100         Statement separator (\) missing
                                        between A+D and 'GOTO'.
0020 IF Q=50 ELSE X=10 \ PRINT Y        The 'THEN' clause is missing.
0030 IF I>1 THEN 200 \ I = I+1          Statement I = I+1 will never
                                        be executed. Error undetected by
                                        BASIC.
```

==========

9.20 IFEND Statement

1 IFEND

See also: ELSE, IF, and THEN statements

Purpose:

The IFEND statement marks the end of a multi-line IF-THEN-ELSE structure.

Comment:

The IFEND should not be part of a THEN or ELSE statement.

The IFEND statement can only be used in conjunction with a multi-line IF statement.

The IFEND statement closes off the corresponding IF statement, marking the end of the conditionally executed statements of the THEN and ELSE statements.

Examples:	Explanation:
0010 IF A	Test A for non zero
0020 THEN GOSUB 2000	Perform if A<>0
0030 PRINT USING "###",A	" " "
0040 GOTO TOP.OF.PAGE	" " "
0050 IFEND	End of conditional execution
0010 IF VALUE > CONTROL	Test expression
0020 THEN IF VALUE > LIMIT	Perform if expr is true
0030 THEN GOSUB ERROR	Perform if both expr are true
0040 GOTO EXIT	" " " " " "
0050 ELSE IF ERR.NUM < ERR.LIMIT THEN QUIT	Perform only if first expr is true and second expr is false
0060 IFEND	End conditional execution from second expr
0070 IFEND	End of conditional execution

9.21 INPUT Statement

```
1 INPUT <variable list>

2 INPUT <prompt>,<variable list>

3 INPUT #<channel>: <variable list>

4 INPUT #<channel>,<key>: <variable list>

Where:

     <channel> ::= <numeric expression>
     <prompt> ::= <string literal expression>
     <variable list> ::= <numeric variable>[,<variable list>]
                         <string variable>[,<variable list>]
     <key> ::= <string expression>
               <numeric expression>

See also: CLOSE, LINPUT, MAT INPUT, MAT READ, OPEN, OPTION, READ,
          and READNEXT statements
```

Purpose:

The INPUT statement allows data to be entered through the console, device, or disk file during program execution.

Comment:

The various formats of the INPUT statement provide different capabilities with one function in common: input fields are always ASCII characters even when the input field is numeric.

Format 1 of the INPUT statement accepts one or more fields of data from the console terminal device.

Format 2 of the INPUT statement accepts one or more fields of data from the console terminal device after displaying the prompting message.

Format 3 of the INPUT statement accepts one or more fields of data from a sequentially accessed device or disk file.

Format 4 of the INPUT statement accepts one or more fields of data from a device or disk file with either direct, indexed, or keyed access.

Format 3 and 4 of the INPUT statement may only be used when the I/O channel has been opened with INPUT or UPDATE access, not OUTPUT.

The <prompt>, when used, must be a string literal expression. That is, the string expression must start with a string literal. When the <prompt> is used the system will evaluate the expression and display the result at the current cursor location followed by the prompt character(s) (see OPTION statement). When the <prompt> is not used, (format 1), the prompt character(s) will be displayed at the current cursor position.

The <variable list> is the list of variables that the input is to be assigned to. This list may be as long as the line allows and may contain a mixture of variable types (numeric, integer, string, array).

When more than one variable is to be entered, each element of data entered must be separated by a comma from the previous element. (Note: when OPTION COMMA is in effect the elements must be separated with a semicolon character.)

When fewer data fields are entered than requested by the list of a format 1 or 2 INPUT, an "Insufficient data" message will be displayed and all data must be re-entered from the beginning of the list. When fewer data fields are entered than requested by the list of a format 3 or 4 INPUT, an "Insufficient data" error occurs (trappable) and execution stops if no ON ERROR is defined.

Input is terminated with a <CR> or end of record indicator.

When using format 1 or 2 and the first character of the first field is a control

character (ASCII value less than 32), the input will be terminated immediately and the value of the control character will be saved in the INP function. Refer to the INP function and the User Definable Keys for more information in this situation.

When the input characters are not enclosed in quotation marks, leading and trailing spaces will be ignored, and embedded commas will be treated as field separators.

The error "Illegal number" will occur when the input variable is numeric and the operator (or file) inputs a non-numeric entry. The system will stop execution if no ON ERROR is defined.

The Line Cancel-key and the backspace key may be used to make corrections to the data being input from the console with INPUT formats 1 and 2.

When the BASIC program was executed from an EXEC program and data was placed in the EXEC Stack, these statements, along with other BASIC statements that accept information from the system console, retrieves the next element from that EXEC Stack. When the EXEC Stack is empty the data must come from the console.

Examples:	Explanation:
0010 INPUT N	A question mark, space is displayed on the terminal, the program supends execution until the operator types a return or enters a control character only.
0020 INPUT "NAME",CUST.NAME$,A,B	The prompt NAME? is displayed and three fields are accepted, one string and two numeric.
0030 OPEN #1: "CONSOLE",INPUT SEQUENTIAL 0040 INPUT #1: A$,B,C	Again, three fields are accepted from the operator, one string and two numeric. No prompt will be displayed and no control characters are allowed.
0050 OPEN #2: "DATE.FILE",UPDATE DIRECT 0060 INPUT #2,13:RECORD$	The 13th record in the file is read into the variable RECORD$.

Incorrect examples:	Explanation:
0020 INPUT A,B,2.3,D	Only variables may be INPUT.
0030 INPUT	At least one variable must be specified.
0040 INPUT "FLD1",F1,"FLD2",F2	Only one prompt is allowed.
0050 OPEN #1: "DATA.FILE",INPUT DIRECT 0060 INPUT #1,"ABCDE": A$,B$	Must use numeric key for direct access.
0070 OPEN #2: "DATA.FILE2",OUTPUT INDEXED 0080 INPUT #2,"ABCDE": A$,B$	Access must be INPUT, not OUTPUT.

9.22 LET Statement

1 [LET] <numeric variable> = <numeric expression>

2 [LET] <string variable> = <string expression>

3 [LET] <string variable><substring> = <string expression>

4 [LET] <user defined function> = <expression>

5 [LET] ERR = <numeric expression>

Where:

<substring> ::= [<numeric expression>:<numeric expression>]

Purpose:

The LET statement assigns a value to a variable.

Comment:

This is the only statement where the statement verb (LET) is not required for proper syntax.

For all of the forms of the LET statement the expression is evaluated and assigned to the element on the left of the first equal sign (may be more than one equal sign because of relational expressions). The previous contents of the element are lost but only after the expression has been evaluated. Therefore, the variable may be an element in the expression.

Formats 1 and 2 of the LET statement are the standard forms of the assignment statement used by all BASIC implementations. The type of the expression (string or arithmetic) must match the type of the variable on the left side of the assignment operator.

The third format of the LET statement provides a powerful method of modifying string variables by means of the substring operator. The general form of this substring is:

<string variable>[<from>:<to>]

Replacement When <from> is less than or equal to <to> a character replacement is performed on the variable from column <from> to column <to>. The string expression on the right side of the assignment operator will be padded with spaces or truncated to a length of <to> minus <from> plus one.

Deletion When <from> is greater than <to> a character deletion is performed on the string variable. The contents of the variable on the left side of the assignment operator is first modified by deleting the characters from, but not including, column <to> through column <from>. The string expression on the right side of the assignment operator is then inserted into the variable after the <to> column.

Insertion When the <from> is zero the string expression on the right side of the assignment operator is inserted after the <to> character position.

These rules and operations are best explained by example:

Assume that A$ contains ABCDEFGHIJ

A$[4:6] = "123"	ABC123GHIJ
A$[6:4] = ""	ABCDGHIJ
A$[6:4] = "01234"	ABCD01234GHIJ
A$[0:6] = "0123"	ABCDEF0123GHIJ
A$[0:0] = "0123456"	0123456ABCDEFGHIJ

Format 4 of the LET statement is the user defined function assignment statement and may only be used within a multi-line user defined function and the function name used on the left side of the assignment operator must be the same as the DEF statement of the user defined function that the LET is a part of. For an example see the DEF and FNEND statements.

Format 5 of the LET statement provides the capability of testing error handling routines during the development phase of a program. This format of the LET allows you to assign a value to the ERR function (error number). When this statement is executed the system will act exactly like it would act if an error occurred. The type of error is determined by the value of the numeric expression on the right side of the assignment operator.

It is advised that this format of the LET statement, when used, should be used in a multi-line statement on the same line as the statement that might cause the same error. For example, use ERR=30 on the same line as an OPEN statement. This is advised because there is no method of setting the ERL function to have a different value than the value of the line number that the LET statement is on.

Examples:	Explanation:
0005 LET A = 1.23	The constant 1.23 is assigned to the variable A.
0010 A = 1.23	Same as previous example.
0020 LET A = A+1	The current value of the variable A is incremented by 1.
0030 LET A$ = "ABCDEF"	The string variable A$ is assigned the ASCII string 'ABCDEF'.
0040 LET A$ = A$&"GHIJ"	The string variable A$ is concatenated with the string 'GHIJ' and afterwards will contain 'ABCDEFGHIJ'.
0050 ERR = 30 OPEN #1: F$,INPUT SEQUENTIAL	Your error handling routine is invoked, if an ON ERROR statement has been used. When the routine is entered ERR = 30 and ERL = 50.

Incorrect Examples:	Explanation:
0010 LET A$ = B+2	Cannot mix string and numeric expressions.
0020 LET B-A = C	Must have a single variable on the left of the assignment operator.
0030 LET +1 = A	Cannot assign a value to a constant.

9.23 LINK Statement

1 LINK <program name>

Where:

<program name> ::= <file name>[.<file type>][:<file disk>]
<file type> ::= BASICOBJ (with BASIC)
BASICCOM (with RUN)

See also: CHAIN, CLEAR, LINK and RUN statements

Purpose:

The primary use of the LINK statement is to link together the segments of a BASIC program.

Comment:

The LINK statement terminates the execution of the program in which it is encountered, loads the program indicated, and continues execution at the beginning of the program segment.

The LINK statement does not close any files, however, all variables that have not been defined as COMMON variables will be cleared from memory.

Previous versions of OASIS BASIC supported the <line number> operand. The recommended method of transferring control to another program at a specific line is the use of a control variable (defined as COMMON) that is tested by an ON-GOTO statement at the start of the program transferred to.

<Program name> is a string expression. If <program-name> cannot be found in the directory, a non-trappable error will occur.

The CHAIN, RUN, and LINK statements all perform similar tasks, but with significant differences:

Program Linkage Statements

Statement	I/O Channels	Variables	COMMON
RUN	Closed	Cleared	Cleared
CHAIN	Closed	Cleared	Not cleared
LINK	Not closed	Cleared	Not cleared

Examples:	Explanation:
0010 LINK "SEGM01"	Program named 'SEGM01' will be loaded and execution resumes at the first line.
0020 LINK NAME$(INDEX%)	The program indicated by the contents of the string array variable NAME$, subscript INDEX will be loaded and execution will resume at the first line.
0030 LINK "SEGM0"&NUM(I)&":S"	When I equals 1, this statement is the same as example 10. When I equals 3 program 'SEGM03' will be executed, etc.

Incorrect example:	Explanation:
0010 LINK "SEGMENT-1"	Program name can only be 8 characters long. Also, - is invalid.

9.24 LINPUT Statement

1 LINPUT <string variable>

2 LINPUT <prompt>, <string variable>

3 LINPUT #<channel>: <string variable>

4 LINPUT #<channel>,<key>: <string variable>

Where:

```
<channel> ::= <numeric expression>
<prompt> ::= <string literal expression>
<key> ::= <numeric expression>
          <string expression>
```

See also: INPUT, LINPUT USING, MAT INPUT, MAT READ, READ, and READNEXT state

Purpose:

The LINPUT statement allows entry of an entire line of data as a single character string, including spaces and punctuation.

Comment:

The LINPUT statement operates identically to the INPUT statement with one exception: only one variable may be specified for input and the variable must be a string variable.

Examples:	Explanation:
0010 LINPUT A$	Prompt character(s) is displayed at the current cursor position, program execution is suspended until the operator terminates the input.
0020 LINPUT "NAME",A$	The literal "NAME? " is displayed, execution is suspended until the operator terminates input.
0030 OPEN #1: "CONSOLE", INPUT SEQUENTIAL 0040 LINPUT #1: STRING$	Similar to line 10 but no prompt nor INP capabilities.
0050 OPEN #2: "DATA.FILE", INPUT DIRECT 0060 LINPUT #1,5: RECORD$	Record number 5 of file is read into variable RECORD$

Incorrect Examples:	Explanation:
0020 LINPUT A1	Must be a string variable.
0030 LINPUT A$,B$,C$	Only one variable is allowed.

9.25 LINPUT USING Statement

1 LINPUT USING <string literal expression>,<string variable>

2 LINPUT <prompt>,USING <string literal expression>,<string variable>

3 LINPUT USING <string expression>,<string variable>

4 LINPUT <prompt>,USING <string expression>,<string variable>

Where:

<prompt> ::= <string expression>

See also: INPUT, LINPUT, MAT INPUT, MAT READ, READ, and READNEXT statements

Purpose:

The LINPUT USING statement allows entry of an entire line of data from the console as a single character string, including spaces and punctuation, with length control and the ability to "modify" an existing field.

Comment:

The LINPUT USING statement, similar to the LINPUT statement discussed previously, allows entry of an entire line of text, including any embedded quotes and commas, as one string field. Similarly, the LINPUT USING allows a prompting message to be displayed before accepting input.

However, unlike the LINPUT statement, the LINPUT USING statement provides greater control of the terminal display by limiting the number of characters input. The most significant feature of the LINPUT USING statement is that the operator can make corrections to the line being entered without re-entering the entire line.

Formats 1 and 2 of the LINPUT USING statement use a <string literal expression> as the using mask. A <string literal expression> is a string expression that starts with a string literal. For example: ""&SPACE$(10) is a string literal expression of length 10.

With either of these formats the statement will display the prompting message if specified. The input area is the area from the starting position for a length specified by the length of the string literal expression.

If the first character of the string literal expression is an exclamation mark (!), BASIC will perform an auto carriage return when the input area if filled. This is generally used on single character input lengths.

Formats 3 and 4 of the LINPUT USING statement use a <string expression> as the using mask. With either of these formats the statement will display the prompting message if specified, then display the string expression in the input area. The input area is the area from the starting position for a length specified by the length of the string expression. Additionally, these formats of the LINPUT USING statement will copy the string expression into the input variable before accepting input.

At this point all four formats of the LINPUT USING statement act the same with the difference being that formats 3 and 4 have pre-filled the input area with the using mask and formats 1 and 2 have a null string in the input area. The operator may enter any ASCII character into the input area. A carriage return will cause the contents of the variable to be saved and execution of the program resumes.

Certain keys are available to the operator to make editing changes:

<carriage return> Terminates entry.

<right arrow> Is a non-destructive advance. When this key is entered (or its equivalent: CTRL/F) the cursor will be advanced over the next character.

<left arrow> Is a non-destructive back space. When this key is entered (or its equivalent: CTRL/H) the cursor will be backed over the current character.

<rub out> Is a destructive back space. When this key is entered the cursor is backed up one position and that character is replaced with a space.

<CTRL/D> Is a destructive delete. When this key is entered the current character is deleted and the remaining characters in the input area are shifted one character to the left.

<CTRL/I> Is a "destructive" insert. When this key is entered (or its equivalent: <tab>) the remaining characters in the input area are shifted one character to the right and a space character is inserted at the current position. If a character shifting to the right would exceed the input area length it will be deleted.

Any other character entered by the operator will replace the current character.

Unfortunately, this statement is difficult to illustrate with a printed example. Therefore, the following program is provided for you to execute so that you may see its uses. Keep in mind that what you see on the terminal in the input area is what is actually in the field being entered.

Any control character (ASCII value less than 32) will terminate the entry and will set the INP function to that value. This implies that the control characters 4, 6, 8, 9, 13 cannot be used as user defined control keys from a LINPUT USING statement. This does not apply to the INPUT or LINPUT statements.

```
0010 OPTION PROMPT "", CASE "M"
0020 PRINT CRT$("CLEAR")
0030 PRINT "The following is a simple illustration of the LINPUT USING"
0040 PRINT "statement in both of its primary forms. The first input request"
0050 PRINT "will use the statement with a string literal expression of"
0060 PRINT "length 30. The second input request will use the statement with"
0070 PRINT "a string expression of length 30. The contents of the string"
0080 PRINT "expression will be the field entered by the first input request,"
0090 PRINT "padded to the proper length."
0100 OPTION CASE "M"  PRINT AT$(1,10);CRT$("EOS");
0110 PRINT AT$(1,10);"Input 1: [";SPACE$(30);"]";AT$(11,10);
0120 LINPUT USING ""&SPACE$(30),FIELD$
0130 PRINT AT$(1,12);"Input 2: [";SPACE$(30);"]";AT$(11,12);
0140 LINPUT USING RPAD$(FIELD$,30),FIELD$
0150 PRINT AT$(1,14);"The field you entered contains:"
0160 PRINT "         |";FIELD$;"|";
0170 OPTION CASE "U"
0180 LINPUT ""&AT$(1,16)&"Okay to repeat (Y/N)? N"&CRT$("L"),USING "!",ANSWER$
0190 IF ANSWER$="Y" THEN 100 ELSE END
```

9.26 MAT Statement

1 MAT <array name> = <array name>

2 MAT <array name> = (<expression>)

See also: LET statement

Purpose:

The MAT statement allows you to either copy one array to another or to assign one value to all of the elements of an array.

Comment:

Format 1 of the MAT statement copies one array to another. Both arrays must have the same dimensions or a "Subscript Range" error will occur.

Forrmat 2 of the MAT statement sets all elements of the array to a specific value.

Example:

```
0010 DIM A$(5),B$(5),C(20)
0020 FOR I%=0 TO 5
0030       B$(I%) = STR(I%)
0040       NEXT
0050 MAT A$ = B$
0060 MAT B$ = ("")
0070 MAT C = (1)
```

Explanation:

```
Define size of arrays A$, B$, and C
Set array B$ to initial values


Copies B$ into A$ (B$ unchanged)
Sets all 6 elements in B$ to be empty.
Sets all 21 elements of C to be 1.
```

Incorrect Example:

```
0010 DIM A$(5),B$(6),C$(5,2)
0020 MAT A$ = (1)
0030 MAT A$ = B$
0040 MAT B$ = C$
```

Explanation:

```
Defines size of arrays A$, B$, and C$.
Expression must match array in type.
Arrays are of different size
  "     "          "     "
```

9.27 MAT INPUT Statement

1 MAT INPUT <array name>

2 MAT INPUT #<channel>: <array name>

3 MAT INPUT #<channel>,<key>: <array name>

Where:

<channel> ::= <numeric expression>
<key> ::= <numeric expression>
<string expression>

See also: COMMON, DIM, INPUT, LINPUT, LINPUT USING, MAT READ, and READ statements

Purpose:

The MAT INPUT statement allows an entire array to be input at one time.

Comment:

Format 1 of the MAT INPUT statement accepts input from the console, assigning each field input to the elements of the array specified. If fewer fields are entered than the remaining elements in the array will be set to zero or null, depending upon the type of the array. The zero subscript of the array will not be input to.

Format 2 of the MAT INPUT statement is identical to format 1 except that the input comes from the file specified by the I/O channel.

Format 3 of the MAT INPUT statement accepts ASCII input from a direct, indexed, or keyed data file. A numeric key must be used if the I/O channel has been opened with access method DIRECT. A string key must be used if the I/O channel has been opened with access method INDEXED. If the wrong type of key is used an "Invalid Key" error will occur.

Formats 2 and 3 may only be used if the I/O channel was opened with access method INPUT or UPDATE. If the channel was opened with access method OUTPUT a "Wrong Access" error will occur.

It is important to note that only one record will be input. If there are fewer fields in the record than there are data elements in the array the remaining elements will be set to zero or null. Zero subscripts will never be input to.

Examples:	Explanation:
0010 OPTION BASE 1 0020 DIM ARRAY(4) 0030 MAT INPUT ARRAY	Accept 4 fields from console
0040 INPUT ARRAY(1),ARRAY(2),ARRAY(3),ARRAY(4)	This statement is identical in function to line 30

9.28 MAT PRINT Statement

```
1 MAT PRINT <array name list>

2 MAT PRINT #<channel>: <array name list>

3 MAT PRINT #<channel>,<key>: <array name list>

Where:

     <array name list> ::= <array name><punct>[,<array name list>]
     <channel> ::= <numeric expression>
     <key> ::= <numeric expression>
               <string expression>
     <punct> ::= <comma>
                 <semicolon>

See also: COMMON, DIM, MAT WRITE, PRINT, PRINT USING, and WRITE
          statements
```

Purpose:

The MAT PRINT statement allows an entire array or arrays to be output at one time.

Comment:

Format 1 of the MAT PRINT statement outputs the arrays to the console.

Format 2 of the MAT PRINT statement outputs the arrays to the file designated by <channel> that was opened for SEQUENTIAL access method.

Format 3 of the MAT PRINT statement outputs the arrays to the file designated by <channel> that was opened for DIRECT, INDEXED, or KEYED access method. A numeric key is required for DIRECT, a string key is required for INDEXED and KEYED. Using the wrong type of key will result in a "Wrong Access" error. This format of the print statement outputs only one record containing all of the elements in the arrays that will fit with the files allocated record length.

Formats 1 and 2 of the MAT PRINT statement may output multiple records. In these formats, the number of records output depends upon the number of dimensions of each array and the number or arrays specified in the list. Additionally, the punctuation character used may cause additional records to be output.

A comma character afer an array name indicates that the array is to be output using print zones, similar to the PRINT statement. A simicolon character after an array name indicates that the array is to be output in a "packed" format, similar to the PRINT statement. When no punctuation is used the array will be output one element per record.

When outputting two dimension arrays using format 1 or 2 of the MAT PRINT statement the second dimension varies fastest. A new record (line) will be started when the first dimension changes.

When multiple arrays are specified a new record will be started for each array. Again, this applies only to format 1 and 2 of the statement.

The zero subscripts of an array are never output with this statement.

Examples:

```
0010 DIM A(5),B(3,10)
0020 FOR I% = 1 TO 5  A(I%) = I%  NEXT I
0030 MAT PRINT A;
0040 PRINT A(1);A(2);A(3);A(4);A(5)
0050 PRINT
0060 MAT B = (1)
0070 MAT PRINT B;
-RUN

 1  2  3  4  5
 1  2  3  4  5

 1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1
```

Explanation:

Statement is identical to line 30

Initializes array B

Output from line 30
Output from line 40

Elements B(1,1) - B(1,10)
Elements B(2,1) - B(2,10)
Elements B(3,1) - B(3,10)

9.29 MAT READ Statement

1 MAT READ <array name>

2 MAT READ #<channel>: <array name>

3 MAT READ #<channel>,<key>: <array name>

Where:

<channel> ::= <numeric expression>
<key> ::= <numeric expression>
<string expression>

See also: COMMON, DIM, INPUT, LINPUT, LINPUT USING, READ, and READNEXT statements

Purpose:

The MAT READ statement allows an entire array to be read at one time.

Comment:

Format 1 of the MAT READ statement accepts data from the DATA statements in the program. If there are fewer DATA elements remaining than there are elements in the array an "Out of Data" error occurs.

Format 2 of the MAT READ statement accepts data from the file opened on the I/O channel specified. Only one record will be read. If there are fewer data elements in that record than there are elements in the array the remaining elements will be set to zero or null, depending upon their type.

The data file used by the second format of the MAT READ statement must have been opened with access method SEQUENTIAL.

Format 3 of the MAT READ statement accepts input from a direct, indexed, or keyed data file. A numeric key must be used if the I/O channel has been opened with access method DIRECT. A string key must be used if the I/O channel has been opened with access method INDEXED or KEYED. If the wrong type of key is used an "Invalid Key" error will occur.

Formats 2 and 3 may only be used if the I/O channel was opened with access method INPUT or UPDATE. If the channel was opened with access method OUTPUT a "Wrong Access" error will occur.

It is important to note that only one record will be input. If there are fewer fields in the record than there are data elements in the array another record will not be read automatically--the remaining array elements will be set to zero or null, depending upon the array type. Additionally, the zero subscript will not be read into.

Although there is not MAT READNEXT statement available performing a READNEXT followed by a MAT READ of the same key will perform the same function.

Examples:

```
0010 OPTION BASE 1
0020 DIM ARRAY(4),A$(2,5)
0030 MAT READ ARRAY                          Accept 4 fields from DATA statement

0040 READ ARRAY(1),ARRAY(2),ARRAY(3),ARRAY(4) Same function as above
0050 DATA 1.23,45,123456788,12335345645E^23
0060 OPEN #1: "DATA.FILE", INPUT DIRECT
0070 MAT READ #1,1: A$                       Ten elements will be read from the
                                             first record in DATA.FILE

0080 READ #1,1: A$(1,1),A$(1,2),A$(1,3),A$(1,4),A$(1,5),A$(2,1),
     A$(2,2),A$(2,3),A$(2,4),A$(2,5)         This statement is identical to line 70.
```

9.30 MAT WRITE Statement

1 MAT WRITE #<channel>: <array name>

2 MAT WRITE #<channel>,<key>: <array name>

Where:

<channel> ::= <numeric expression>
<key> ::= <numeric expression>
<string expression>

See also: COMMON, DIM, MAT PRINT, PRINT, PRINT USING, and WRITE statements

Purpose:

The MAT WRITE statement allows an entire array to be output at one time.

Comment:

Format 1 of the MAT WRITE statement outputs data to the file opened on the I/O channel specified. Only one record will be output.

The data file used by this format of the MAT WRITE statement must have been opened with access method SEQUENTIAL.

Format 2 of the MAT WRITE statement outputs data to a direct, indexed, or keyed data file. A numeric key must be used if the I/O channel has been opened with access method DIRECT. A string key must be used if the I/O channel has been opened with access method INDEXED or KEYED. If the wrong type of key is used an "Invalid Key" error will occur.

Formats 1 and 2 may only be used if the I/O channel was opened with access method OUTPUT or UPDATE. If the channel was opened with access method INPUT a "Wrong Access" error will occur.

It is important to note that only one record will be written. If there is insufficient space allocated for the record it will be truncated.

The zero subscript of the array will never be written.

Examples:

```
0010 OPTION BASE 1
0015 OPEN #1: "TEST.FILE:A", OUTPUT SEQUENTIAL, EXTEND
0020 DIM ARRAY(4),A$(2,5)
0030 MAT WRITE #1: ARRAY
```

Outputs 4 fields to the file on channel 1

```
0040 WRITE #1: ARRAY(1),ARRAY(2),ARRAY(3),ARRAY(4)
```

This statement is identical in function to line 30

```
0060 OPEN #2: "DATA.FILE", INPUT DIRECT
0070 MAT WRITE #2,1: A$
```

Ten elements will be written to the first record in DATA.FILE

```
0080 WRITE #2,1: A$(1,1),A$(1,2),A$(1,3),A$(1,4),A$(1,5),A$(2,1),
     A$(2,2),A$(2,3),A$(2,4),A$(2,5)
```

This statement is identical to line 70.

9.31 MOUNT Statement

1 MOUNT <string exp>

See also: CLOSE statement

Purpose:

The MOUNT statement allows the operator to change a disk without returning to the operating system.

Comment:

The MOUNT statement may only be used to change a privately owned disk. (In single user OASIS all disks are privately owned.)

When OASIS is in BASIC, or any program, a record is kept of the disk labels and in which drive these disks are loaded. By doing this the operating system is protecting the user from inadvertantly changing disks without the express permission of the program being given. In BASIC this permission is given by the MOUNT statement.

The MOUNT statement instructs the operating system that the program is prepared for a change of disk. No messages are displayed by the operating system at this time: any prompting messages to the operator must be handled by the BASIC program.

The string expression specifies which drive is to be mounted (A, B, C, etc.).

When BASIC executes the MOUNT statement a check is made to insure that there are no open files on the specified disk. If there are any open files the statement is not executed and an error message is displayed: 'File Error at Line nnnn'.

After the MOUNT statement has been executed the disk may be changed (with the exception of the system disk).

Example:	Explanation:
0010 MOUNT "A"	System checks for any open files on disk A. If no files are open then pointers are set to indicate that the disk may be changed.

9.32 NEXT Statement

1 NEXT <variable>

2 NEXT

See also: FOR statement

Purpose:

The NEXT statement marks the outer limit of control of a FOR statement and causes the loop to be repeated if the limit has not been reached.

Comment:

The variable, if specified, must be the same variable in use as an index variable for a currently open FOR loop. When the variable is not specified the current FOR index variable is used.

When the NEXT statement is executed control of the program returns to the FOR statement indicated, at which time the index variable will get its next value, the value will be tested against the limit, and the program will continue depending upon the result of the test.

An attempt to execute a NEXT statement when no FOR loop is open (or the variable specified does not match any FOR loop that is open) will cause an error to occur: "NEXT without FOR".

Caution: Format 2 of the NEXT statement should not be used if it is possible that another, unfinished FOR-NEXT loop might be in existence--control will be transferred to that other unfinished loop. This situation is difficult to debug when it occurs.

Examples:	Explanation:
0010 FOR I%=1 TO 5 0020 PRINT I% 0030 NEXT I%	Repeats following instructions five times. This marks the end and causes repeat
0060 FOR I%=1 TO 5 STEP 1 0070 PRINT I% 0080 NEXT	Same as above.
0100 FOR I$="A","B","C" 0110 PRINT I$ 0120 FOR I%=1 TO 5 0130 PRINT I% 0140 NEXT I% 0150 NEXT	Performs loop 3 times. Performs this loop 5 times for each of the 3 major loops. Marks end and repeat of sub-loop. Marks end and repeat of major loop.

9.33 ON ERROR Statement

```
1 ON ERROR GOTO <line reference>

2 ON ERROR GOTO 0

Where:

     <line reference> ::= <line number>
                          <line label>

See also: RESUME statement
```

Purpose:

The ON ERROR statement allows you to specify the error subroutine to be used for trappable errors.

Comment:

Normally BASIC detects an error while executing a program and either terminates execution or prints a warning message. However, if you plan ahead, you can prepare alternatives which can save you time in the event of an error (and avoids confusion on the part of the operator). You can build an error handling routine that is activated when, and if, BASIC finds an error. This routine takes control away from the normal system errors and gives it to your error handling routine.

The ON ERROR statement instructs BASIC that a user error handling routine exists at a certain line or that the currently defined error handling routine is to be disabled.

Format 1 indicates that the specified line is to receive control; format 2 indicates that BASIC is to handle all errors.

When an error occurs before the execution of a format 1 ON ERROR statement or after the execution of a format 2 ON ERROR statement, BASIC proceeds with normal system error handling.

When format 1 of the ON ERROR statement is executed and a trappable error occurs, control will be transferred to the line specified. That line should be the start of your error handling routine.

An error handling routine can make decisions about how to handle the error by interpreting the error functions ERR and ERL which return the number of the error and the line number that the error occurred on.

Note: When an error handling routine is being executed errors will not be trapped.

Note: An error handling routine will always be disabled by RUN, CHAIN, or LINK--each segment must redefine the error handling routine.

An error handling routine may be tested by using one of the formats of the LET statement to invoke the error routine.

The ON ERROR statement may be used within an error handling routine.

The error "ESC-C" is not trapped while interpretive BASIC is running a program--only during execution of a compiled program. This allows the programmer to stop a program while he is still developing it.

Example:

```
0010 ON ERROR GOTO ERROR.ROUTINE
 .
 .
9010 ERROR.ROUTINE:
9020      SELECT ERR
9030           CASE 1
9040                IF ERL<1000 OR ERL>1999 THEN RESUME Ignore if not of interest
9050                GOSUB CLOSE.REPORT else do this.
9060                RESUME MENU
9070           CASE 20
9080                IF ERL=990 THEN RESUME 991
9090                RESUME
9100           CASE 30
9110                PRINT AT$(1,24);"Invalid file name";CHR$(7);
9120                LINPUT " Type <return> to continue: ",USING "!",ANSWER$
9130                RESUME
9140           CEND
9150      RESUME 0
```

Explanation:

Trappable errors will be handled by user.

Using error functin, select error routine.
Perform if ERR=1 (escape,C)
Ignore if not of interest
else do this.
Perform if ERR=20 (on range)
Perform if ERR=30 (file not found)

9.34 ON GOSUB and ON GOTO Statements

1 ON <num expr> GOTO <line list>

2 ON <num expr> GOSUB <line list>

Where:

```
<line list> ::= <statement reference>[,<line list>]
<statement reference> ::= <line number>
                          <line label>
```

See also: ON ERROR, GOTO, GOSUB and RETURN statements

Purpose:

These statements transfer control to a line selected from a list by the integer value of an expression.

Comment:

The keywords GOTO and GOSUB may be entered as GO TO and GO SUB.

The expression following ON is evaluated and the value is integerized. The integer is then used to select the first, second, third, etc., line reference. A trappable error occurs when the value of the integer is less than or equal to zero, or the value of the integer is greater than the number of line references.

The subroutine given control by an ON GOSUB statement should be exited only with a RETURN statement. When the RETURN statement is executed, control returns to the statement following the ON GOSUB statement. This is further explained in the descriptions of the GOSUB and RETURN statements.

The line references following GOTO or GOSUB must be separated by commas or spaces.

There may be any number of line references in the list, (limit of 255 characters per line).

Line references may be omitted by using the comma separator as a place holder. When this is done and the value of the expression corresponds to that line reference place then control will be transferred to the statement following the ON statement.

If the value of the <numeric expression> is less than one or greater than then number of line references in the <line list> an "ON range" error will occur.

Examples:	Explanation:
0010 ON I GOTO 100,110,120,130	When I=1 control passes to line 100, I=2 then line 110, I=3 then line 120, I=4 then line 130.
0020 ON I+1 GOTO 100,120	When I=0 control passes to line 100, I=2 then line 120.
0030 ON I GOSUB 100,200,300	When I=1 the subroutine starting at line 100 is executed, I=2 the subroutine at 200 is executed, etc.
0040 ON INDEX+4 GOTO LINE1,,LINE3	When INDEX=-3 control passes to LINE1; INDEX=-2 control passes to line following this; INDEX=-1 control passes to LINE3. All other values of INDEX will cause an error to occur.

Incorrect example:	Explanation:
0020 ON I$ GOTO 100,200,300	Expression must be numeric.

9.35 OPEN Statement

```
1 OPEN #<channel>: <file>,<access mode> <access method>[,<options>]

2 OPEN #<channel>: <device>,<access mode> <access method>[,<options>]

3 OPEN #<channel>: <null>,<access mode> <access method>[,<options>]

Where:

     <channel> ::= <numeric expression>
     <file> ::= <string expression>
     <device> ::= <string expression>
     <access mode> ::= INPUT
                       OUTPUT
                       UPDATE
     <access method> ::= SEQUENTIAL
                         DIRECT
                         INDEXED
                         KEYED
     <options> ::= <option>[,<options>]
     <option> ::= EXTEND
                  QUOTE
                  FORMAT
                  LOCK
     <number> ::= <integer constant>

See also: CLOSE and UNLOCK statements
```

Purpose:

The OPEN statement provides you with the initial means of accessing I/O devices other than the console.

Comment:

<channel> must be a number with a value between 1 and 16. This number is the channel number that the file is assigned to.

<file> represents the file description of the disk file to be opened.

<device> represents the device name of the device to be opened. The table at the end of this section defines the allowable device names. The device name must be spelled out (no appreviations or synonyms).

The <file> must include the file name (fn) and file type (ft), but the file disk (fd = A, B, S, etc., or the disk label) is optional. The proper separators must be used: a period before file type and a colon before the file disk. If the file disk is omitted the system will search the directories of the disks attached in the default search sequence.

When the <file> or <device> is a null string (format 3) the channel is opened for the device or file that was last opened on this same channel number. This feature, in conjunction with the ASSIGN command in OASIS, allows you to write BASIC programs that are device independant. For example I/O channel 16 might be used for report files. An ASSIGN command (see OASIS System Reference Manual) might assign channel 16 to PRINTER1. In the BASIC program channel 16 is opened to a null string. BASIC opens PRINTER1 on channel 16.

<access mode> represents an unquoted literal indicating the primary access mode or direction of the device or file:

INPUT indicates that the file or device is to be used as an input source of an existing data base. No record locking will be performed on a file opened for INPUT.

OUTPUT indicates that the file is to be used as an output storage base. No record locking will be performed.

UPDATE indicates that the file is to be used as a general data base for both input and output. Input operations (input and reads) on this file will cause the specific record to be locked. This record will be released by

a subsequent read or write to the file, an UNLOCK of the file, or by closing the file.

<access mode> restricts how a file will be used: INPUT mode only allows the statements INPUT, LINPUT, MAT INPUT, MAT READ, READ, and READNEXT to be executed on the specified channel; OUTPUT mode only allows the statements MAT PRINT, MAT WRITE, PRINT, PRINT USING, and WRITE to be executed on the specified channel; UPDATE mode allows all file access statements to be executed on the specified channel.

<access method> specifies what access method is to be used:

SEQUENTIAL indicates that records will be accessed in a sequential manner, one after the other. This applies to both input and output to the file.

DIRECT indicates that records will be accessed in a random manner by relative number.

INDEXED indicates that records will be accessed in a random manner by key and that the file will be maintained in the ASCII collating sequence of the keys.

KEYED indicates that records will be accessed in a random manner by key.

<access method> also refers to the file format.

Note: a file opened for OUTPUT SEQUENTIAL will erase any existing file with the same description and create a new file.

The specific access method specified in the OPEN statement affects the required syntax of the file access statements. For example, a file opened with access method INDEXED or KEYED will require that all statements accessing that channel use a string key; similarly, access method DIRECT will require that all statements accessing that channel use a numeric key; access method SEQUENTIAL will require that all statements accessing that channel not use a key.

The access mode in combination with the access method have other implications and corresponding requirements:

INPUT SEQUENTIAL implies that the file already exists.

OUTPUT SEQUENTIAL implies that the file is to be created by this program (unless option EXTEND is used).

UPDATE SEQUENTIAL is the same as OUTPUT SEQUENTIAL.

<option> specifies additional functions to be performed by the OPEN statement:

EXTEND indicates that the sequential format file's disk allocation is to be extended.

QUOTE indicates that string fields output with the PRINT statement, as part of a record, are to be surrounded with quotes if the string contains any embedded quotes or commas, or leading or trailing spaces. A comma will always be output between fields.

FORMAT indicates that the SEQUENTIAL access method file is to use ANSI forms control characters supplied by each PRINT or PRINT USING statement.

LOCK indicates that the entire file is to be locked from other users use until the file is closed by this program.

An error occurs when a nonexistent file is opened for mode INPUT or access methods DIRECT, INDEXED, or KEYED.

An error occurs when the channel number is still in use by another file.

When a file is opened for SEQUENTIAL, the record pointer is set to the first record. This is the only statement that sets the record pointer to the beginning of a file opened for access method SEQUENTIAL.

When a disk file is opened for OUTPUT SEQUENTIAL, the file is first erased, and then created. When a disk file is opened for OUTPUT SEQUENTIAL with option EXTEND, and the file exists, the output record pointer will be positioned to the end of the file where records will be added.

A file with delete protection may not be opened for OUTPUT. If this occurs, the

error message "Protected File" will be displayed.

Device/Mode Relationships

access mode / device	INPUT I	OUTPUT O	UPDATE I/O
CONSOLE	X	X	X
PRINTER[n]		X	
TAPE[n]	X	X	
COMM[n]	X	X	X
READER	X		
PUNCH		X	
DUMMY	X	X	X

Disk File format/Mode, Method Relationships

mode / method	sequential	direct	indexed	keyed
INPUT SEQUENTIAL	file must exist			
INPUT DIRECT	N/A	X		
INPUT INDEXED	N/A		X	
INPUT KEYED	N/A			X
OUTPUT SEQUENTIAL	file recreated			
OUTPUT DIRECT	N/A	X		
OUTPUT INDEXED	N/A		X	
OUTPUT KEYED	N/A			X
UPDATE SEQUENTIAL	file recreated			
UPDATE DIRECT	N/A	X		
UPDATE INDEXED	N/A		X	
UPDATE KEYED	N/A			X

Examples:

```
0010 OPEN #1: "MAST.DATA:A",INPUT SEQUENTIAL
0020 OPEN #1: "",INPUT DIRECT
0030 OPEN #5: "TEST.DATA:C",OUTPUT DIRECT
0040 OPEN #I: "PRINTER",OUTPUT SEQUENTIAL,FORMAT
0050 OPEN #8: F$&":S",UPDATE INDEXED,LOCK
0060 OPEN #16: "CONSOLE",OUTPUT SEQUENTIAL
0070 OPEN #15: "PRINTER",OUTPUT SEQUENTIAL
0080 OPEN #4: "PRINTER.FILE:S",OUTPUT SEQUENTIAL,EXTEND,FORMAT
0090 OPEN #16: "CUSTOMER.MASTER",UPDATE KEYED
```

9.36 OPTION Statement

```
1 OPTION BASE <base value>

2 OPTION CASE <case mode string expression>

3 OPTION PROMPT <prompt>

4 OPTION USR <usr-name>

5 OPTION PRIV <privlev>

6 OPTION SERIAL <serial-number>

7 OPTION COMMA

Where:

     <base value> ::= 0 | 1
     <case mode string expression> ::= "M" | "U" | "L"
     <prompt> ::= <string expression>
     <usr-name> ::= <string expression>
     <privlev> ::= 0 | 1 | 2 | 3 | 4 | 5
     <serial-number> ::= <unsigned int 1>-<unsigned int 2>
     <unsigned int 1> ::= 0 - 255
     <unsigned int 2> ::= 0 - 65535

See also: USR function
```

Purpose:

The OPTION statement allows the programmer to specify the status of certain global options: array subscript base value, input casemode, and input prompt character(s).

Comment:

The OPTION BASE statement must be in a position to be executed before any variables are dimensioned or defined. This also means that the subscript base cannot be changed after COMMON has been defined. Normally the OPTION BASE statement would be the first statement of the first segment of the program.

When the OPTION BASE statement is not used the default base is 0.

Since most programmers do not use the zero element of arrays the OPTION BASE 1 allows for a saving in the memory space used for working storage.

More than one option may be specified in an OPTION statement by separating the options with a comma. For example: 10 OPTION BASE 1,PROMPT CHR(0),CASE "M"

The OPTION CASE and OPTION PROMPT statements may be used in any location of the program that a BASIC statement is allowed. The OPTION CASE statement specifies the casemode for characters entered from the console input device (CONIN). When the casemode is not set by the programmer the default mode of upper is used.

OPTION CASE "U" indicates that all alphabetic characters entered from CONIN are to be translated to their uppercase equivalent before display and before the character(s) are transferred to the BASIC program.

OPTION CASE "M" indicates that all alphabetic characters entered from CONIN are not to be translated.

OPTION CASE "L" indicates that all alphabetic characters entered from CONIN are to be translated to their inverse casemode equivalent before display and before the character(s) are transferred to the BASIC program.

When BASIC or RUN is first invoked the casemode of input is "U". In order to use mixed or lowercase characters for input to the BASIC program an OPTION CASE "M" or CASE "L" statement must be executed. This may be done in the immediate mode.

The OPTION PROMPT statement changes the prompt literal. The default prompt literal is the question mark followed by a space. By using this statement you can change the prompt to be any character, or sequence of characters, or you can change the

prompt to be a null string not followed by a space. OPTION PROMPT "" and OPTION PROMPT CHR$(0) are equivalent and indicate that no prompt literal is to be used for INPUT and LINPUT statements to the console.

Format 4 of the OPTION statement loads a USR assembly language subroutine into memory. Note: this is the only way that a USR program is loaded for use by the BASIC program.

When the USR program referenced is already in memory no action will be taken by this statement.

Format 5 of the OPTION statement specifies the lowest privilege level allowed to executed the compiled program. (Privilege level is not checked in the interpretive mode.)

Format 6 of the OPTION statement specifies the serial number of the operating system that may execute the compiled program. (Serial number is not checked in the interpretive mode.)

Note: Formats 5 and 6 of the OPTION statement if used, must be the first executable statement in a program.

Format 7 of the OPTION statement specifies that input and output of numbers in their ASCII format is to conform to the European standard regarding commas and periods. (See chapter "Introduction" in this manual.)

Examples:	Explanation:
0001 OPTION SERIAL 3-12345, PRIV 3	
0010 OPTION BASE 1	Set index base for arrays to 1.
0020 OPTION CASE "M"	Accept input with no translation
0030 OPTION PROMPT CHR$(0)	No prompting character or space.
0040 OPTION PROMPT "Enter:", CASE "L"	The input prompting literal changed to the characters: Enter: followed by a space; the case mode of input set to invert.
1000 CLEAR \ OPTION BASE 1	Array subscript base set to 1.
1010 OPTION PROMPT "What? ",USR "XX"	Prompt literal changed to What? and the USR program named XX is loaded into memory.

9.37 OTHERWISE Statement

1 OTHERWISE

See also: CASE, CEND and SELECT statements

Purpose:

The OTHERWISE statement specified the action to be taken in a SELECT-CASE-CEND structure if none of the previous cases were true.

Comment:

The OTHERWISE statement functions similar to the CASE statement except that there is no expression specified--the OTHERWISE statement is always true.

The OTHERWISE statement allows you to specify an action (sequence of statements) to be executed when none of the cases is true in a SELECT-CASE-CEND programming structure.

There should only be one OTHERWISE statement in any particular SELECT structure (only one will be executed).

The OTHERWISE statement should follow all CASE statements in a SELECT structure (no CASE statements will be evaluated after the OTHERWISE statement is encountered).

Examples:

```
3000 SELECT RAD*2.*PI
3010      CASE 0
3020           SELECT SUBVALUE%          Perform only if RAD*2.*PI=0
3030                CASE 20
3040                  .                  Perform only if RAD*2.*PI=0 and SUBVALUE%=20
3050                CASE 32
3060                  .                  Perform only if RAD*2.*PI=0 and SUBVALUE%=32
                                          = 32
3061                OTHERWISE
3062                  .                  Perform if neither of the above cases
                                          is true
3070                CEND                 End of nested SELECT structure
3080      CASE I-14
3090           .                         Perform only if RAD*2.*PI=I-14
3100           .                              "    "  "   "   "
3110      CASE J%
3120           .                         Perform only if RAD*2.*PI=J%
3130      CEND                           End of SELECT structure
```

9.38 PRINT Statement

1 PRINT

2 PRINT <expression list><punctuation>

3 PRINT #<channel>

4 PRINT #<channel>,<key>

5 PRINT #<channel>:<expression list><punctuation>

6 PRINT #<channel>,<key>:<expression list>

Where:

```
<expression list> ::= <expression>[<punct><expression list>
                      TAB(<num expr>)[<punct><expression list>
<punctuation> ::= <comma>
                  <semi-colon>
<channel> ::= <numeric expression>
<key> ::= <numeric expression>
          <string expression>
```

See also: CLOSE, MAT PRINT, MAT WRITE, OPEN, PRINT USING, and WRITE statements

Purpose:

The PRINT statement allows text, numbers, results, etc., to be displayed on the console or output to a file.

Comment:

The various formats fo the PRINT statement provide different capabilities with one function in common: output is always ASCII, even when the output field is numeric.

Format 1 of the PRINT statement prints a carriage return on the console.

Format 2 of the PRINT statement prints one or more fields of data on the console.

Format 3 of the PRINT statement outputs a null or empty record on the sequentially accessed device or disk file.

Format 4 of the PRINT statement outputs a null or empty record on the direct or indexed accessed disk file.

Format 5 of the PRINT statement outputs one or more fields of data to the sequentially accessed device or disk file.

Format 6 of the PRINT statement outputs one or more fields of data to the direct or indexed accessed disk file.

The PRINT statement, formats 3 through 6, may only be used on an I/O channel that was opened with access mode OUTPUT or UPDATE. An attempt to execute a PRINT statement on a channel opened for INPUT will cause the error "Wrong access" to occur.

The expressions (formats 2, 5, and 6) will be output in the order that they are listed.

Using the expression feature (e.g., 2*R^3-B) can be very valuable in saving programming time, execution time, and memory usage. For instance: if the result of an expression is only calculated in order to be output, and there is no repetition of it's output, it is best to use the expression in the PRINT statement. In this case line 20 below is the more efficient way to code:

```
10 LET A = 2*R^3-B \ PRINT A
20 PRINT 2*R^3-B
```

Both statements will yield the same results.

All numeric expressions (literals, fields, and expressions) will be printed with leading zero suppression, left justification, leading sign or space, and one trailing space.

The term "print head", used bellow, refers to the cursor (terminals), the print mechanism (printers) or the record pointer (disk files), whichever is applicable.

An output record is considered to be divided into print zones of twenty one spaces each. To use these zones for tabulation, the punctuation character is a comma. In the PRINT statement, an expression followed by a comma will cause the value of the expression to be printed at the current print position. After printing, the "print head" will be moved to the next available print zone (from 1 to 21 spaces away). If the last print zone on a line is filled, the "print head" will move to the first print zone of the next line.

In the PRINT statement, an expression followed by a semicolon (;) will cause the value of the expression to be output at the current print position with no movement of the "print head" after printing.

Any PRINT statement which ends with no punctuation causes the "print head" to move to the first column of the next line after output.

Printing to an I/O channel (formats 3, 4, 5, and 6) may require the use of ANSI forms control characters, depending upon whether or not the option FORMAT was used in the open statement for that channel. The FORMAT option should only be used for terminal or printer files. When it is used it means that the PRINT statement will supply the forms control character as the first character of each record output. When it is not used it means that each record output is to start a new line on the output device. For a list of these forms control characters refer to the OASIS System Reference Manual, appendix on "ANSI Forms Control". These characters allow you to specify single, double, triple spacing, forms eject, or no line spacing (overprint).

If the option QUOTE was used on the open statement for the I/O channel used by a PRINT statement, the fields output to the device or file will be enclosed in a pair of quotation marks if the field contains any of the following: leading spaces, trailing spaces, embedded comma, or embedded quotation mark. If it is unknown whether this will happen it is best to use the QUOTE option--no action is taken unless needed. Additionally, the QUOTE option causes multiple fields to be separated by commas in the output record.

The QUOTE option causes punctuation in the expression list to be ignored (commas are used as stated above).

If the QUOTE option is not used then leading and trailing spaces will be removed from the fields before output. Additionally, when a field is output that contains an embedded comma a subsequent INPUT of that record will treat the comma as a field separator, not as an embedded comma (LINPUT will not be concerned with this). Embedded quotes in a field might also cause a problem for the INPUT statement.

In general, the output rules for the PRINT statement are:

1. Suppression of leading and trailing zeros to the right of a decimal point.
2. Where a number can be represented as an integer, printing of the decimal point is suppressed.
3. At most, thirteen significant digits are printed.
4. Most numbers are printed in decimal format. Numbers too large or too small to be printed in decimal format are printed in exponential format.
5. Extra commas cause print zones to be skipped. (Unless option QUOTE is in effect.)
6. A semicolon at the end of the list indicates that no carriage return, line feed is to be printed.
7. Leading and trailing spaces in string expressions are removed (unless option QUOTE is in effect).
8. Numeric fields are output with a leading sign (negative values) or space (positive values) and a trailing space (unless option QUOTE is in effect).

The examples are followed by the printout caused by their execution.

==========

Examples:

```
0010 LET A = 1.23 \ B = 34.56 \ C = 345.678
0020 LET A$ = "ABCDEFG" \ B$ = "HIJKLMN" \ C$ = A$+B$
0035 OPEN #1: "PRINTER",OUTPUT SEQUENTIAL,FORMAT
0030 PRINT #1: " A =";A;"B =";B;"C =";C
0040 PRINT #1: " A =";A;"B =";B;"C =";C
0050 PRINT #1: " ";A+B+C,A*B,C*A,B/A,A/B
0060 PRINT #1: " ";A$;B$;C$;
0070 PRINT #1: "    ";B$;
0080 PRINT #1: " ";A$
0100 OPEN #2: "DATA.FILE:A",OUTPUT DIRECT,QUOTE
0110 PRINT #2,5: A$,B$,C,D,E            Fifth record is output to the file
```

```
A = 1.23 B = 34.56 C = 345.678
A = 1.23 B = 34.56 C = 345.678
 381.467999998      42.5087999999      425.183939999      28.0975609756
 .355902777778
ABCDEFGHIJKLMNABCDEFGHIJKLMN  HIJKLMNABCDEFG
```

Incorrect examples:	Explanation:
0010 PRINT "ABCDEF	Expression illegal.
0020 PRINT A,B:C	Invalid punctuation.

==========

9.39 PRINT USING Statement

```
1 PRINT USING <mask>,<expression list><punctuation>

2 PRINT #<channel>: USING <mask>,<expression list><punctuation>

3 PRINT #<channel>,<key>: USING <mask>,<expression list><punctuation>

Where:

     <mask> ::= <string expression>
     <expression list> ::= <expression>[,<expression list>
     <punctuation> ::= <semi-colon>
     <channel> ::= <numeric expression>
     <key> ::= <numeric expression>
               <string expression>

See also: MAT PRINT, MAT WRITE, PRINT, and WRITE statements
```

Purpose:

The PRINT USING statement allows text, numbers, results, etc., to be displayed on the console or output to a file.

Comment:

The PRINT USING statement operates similar to the PRINT statement except that: fields must be output, output is formatted.

Format 1 of the PRINT USING statement outputs formatted data to the console terminal.

Format 2 of the PRINT USING statement outputs formatted data to a device or disk file opened for SEQUENTIAL access.

Format 3 of the PRINT USING statement outputs formatted data to a disk file opened for DIRECT or INDEXED access.

The PRINT USING statement may only output to an I/O channel opened for OUTPUT or UPDATE access. An attempt to access a channel opened for INPUT will cause the error "Wrong access" to occur.

The expressions will be displayed in the order that they are listed, in the format specified by the mask expression. For details on the mask specifications refer to the chapter on "Formatted Output" in this manual. Expressions can be string or numeric literals, variables, expressions, or functions, as long as they match in type to the formatting masks specification types.

Option QUOTE of the OPEN statement has no effect on the PRINT USING output; however, option FORMAT has the same effect as it does for the PRINT statement.

In the PRINT USING statement, all expressions must be separated by commas. A semicolon is allowable as the terminating punctutation and, if used, operates the same way the semicolon punctuation character operates in the PRINT statement.

A PRINT USING statement which ends with no punctuation causes the print head to move to the first column of the next line after printing.

The examples are followed by the printout caused by their execution.

For examples of the PRINT USING statement and its output capabilities refer to the chapter on "Formatted Output".

The following program example, when entered and executed, will show some of the uses of the PRINT USING statement.

```
0010 OPTION PROMPT "", BASE 1
0020 DIM NUMBER(5),STRING$(5)
0030 OPTION CASE "M"  PRINT CRT$("CLEAR")
0040 PRINT "PRINT USING example program"
0050 LINPUT ""&AT$(1,4)&"Numeric mask: ",MASK$
0060 LINPUT ""&AT$(40,4)&"String mask: ",MASK1$
0070 PRINT
0080 PRINT "Enter five numbers:";TAB(40);"Enter five strings:"
0090 FOR I% = 1 TO 5
0100      PRINT AT$(5,I%+7);I%;
0110      INPUT NUMBER(I%)
0120      NEXT I%
0130 FOR I% = 1 TO 5
0140      PRINT AT$(45,I%+7);I%;
0150      LINPUT STRING$(I%)
0160      NEXT I%
0170 MASK$ = MASK$&" "&MASK1$
0180 PRINT AT$(1,14);"The formatted output of: "&MASK$
0190 PRINT AT$(1,16);
0200 PRINT USING MASK$,NUMBER(1),STRING$(1),NUMBER(2),STRING$(2),NUMBER(3),STR
     ING$(3),NUMBER(4),STRING$(4),NUMBER(5),STRING$(5)
0210 OPTION CASE "U"
0220 LINPUT ""&AT$(1,23)&"Okay to repeat (Y/N)? ", USING "!",ANSWER$
0230 IF ANSWER$="Y" THEN 30
0240 END
```

9.40 PUT Statement

1 PUT DEVICE <device number>,<expression list>

2 PUT MEMORY <address>,<expression list>

3 PUT PORT <port>,<expression list>

Where:

<device number> ::= <numeric expression>
<address> ::= <numeric expression>
<port> ::= <numeric expression>
<expression list> ::= <numeric expression>[,<exp list>]
<string expression>[,<exp list>]

See also: GET and WAIT statements

Purpose:

The PUT statement allows the user to output a single byte or a list of bytes to an I/O device such as an digital to analog (D/A) converter or some other device.

Comment:

<Port>, <device>, and <address> are numeric expressions which are rounded up and integerized. <port> must be in the range: 0 - 255. This number is the address of the I/O port. <device> must be in the range of 9 - 32. This number is the logical device number (see OASIS System Reference Manual).

<address> must be in the range -32767 - 32767. This value, unlike other integer values, is interpreted as an unsigned value, which automatically adjusts the range to 0 - 65535. It is best to use hexadecimal values for <address> as they are more easily interpreted as unsigned values.

The expressions in the expression list are evaluated. If the expression is numeric, it must be in the range 0 - 255. If the expression is a string, only the first byte is used. When more than one expression is specified each is evaluated independently of the others. When PUT MEMORY is used with multiple expressions the memory address is incremented by 1 for each byte transmitted.

BASIC does not test to see if the I/O device is ready before transmitting the byte. This is the responsibility of the user (see WAIT statement).

The PUT statement is identical to the GET statement except that data is output to the logical device driver (DEVICE), port address (PORT), or memory locations (MEMORY), instead of input. When the PUT PORT or PUT MEMORY statements are used you must be careful not to destroy the operating system. It is very easy to do.

Examples:	Explanation:
0010 PUT DEVICE 10,65,66,"C"	On the console output device (number 10), the characters A, B, and C are output.
0020 PUT PORT 1,"A"	The letter "A" is written to port 1.
0030 PUT MEMORY 3000H,0,0,0FFH	At memory locations 3000, 3001, and 3002 hexadecimal, the values 0, 0, and 255 are placed, respectively.

9.41 QUIT Statement

1 QUIT

2 QUIT <string expression>

3 QUIT <numeric expression>

See also: END statement and QUIT command

Purpose:

The QUIT statement allows the user to exit from the BASIC environment.

Comment:

When the QUIT statement is encountered by BASIC all open I/O channels are closed.

The QUIT statement always exits from BASIC. If BASIC (or RUN) was invoked by a keyboard command then control is returned to the Command String Interpreter environment. If BASIC (or RUN) was invoked by an EXECutive procedure then control is returned to the EXECutive procedure that called it. The EXEC resumes control with the statement that followed the BASIC command. In either case the return code is set to zero.

To exit BASIC without returning control directly to the environment that it was invoked from one of the optional expressions is specified.

A numeric expression indicates the value that the return code is to be set to. This return code may then be examined by the EXEC that invoked BASIC. If BASIC was not invoked by an EXEC then setting the return code will have no usable effect.

A string expression indicates a CSI command to be executed. The expression must specify the command name and all arguments and options desired. After the command has completed execution the return code is set by that command. If BASIC was invoked by an EXECutive procedure and a string expression is specified with the QUIT statement control will return to the EXEC program after the CSI command has completed execution.

When the first character of the string expression is the character ">" the string command will be displayed on the console terminal.

Examples:	Explanation:
0900 QUIT	Control exits BASIC
9998 QUIT 3	Return code set to 3; BASIC is exited.
9990 QUIT "LIST DAILY REGISTER"	BASIC is exited and LIST executed.

9.42 RANDOMIZE Statement

1 RANDOMIZE

Purpose:

The RANDOMIZE statement causes the RND function to use a random starting value.

Comment:

The RANDOMIZE statement is used when a program that uses the RND function is to have a different set of random numbers each time the program is run.

The RND function does not produce truly random numbers: it has a "table" of pseudorandom numbers available to it. Using the last random number generated, the RND function chooses another 'random' number. Every time that BASIC is loaded into memory it has the same starting pointer to the pseudorandom number "table". The RANDOMIZE statement causes this pointer to start at a different location each execution of the program.

It is a good practice to debug a program completely before inserting the RANDOMIZE statement.

The RANDOMIZE statement is normally used only once in a program, generally at the beginning of the logic.

Examples:	Explanation:
0010 RANDOM	Choose a random starting point.
0020 PRINT INT(RND*10.)	Print a random number between 0 and 10.

Incorrect examples:	Explanation:
0010 RANDOMISE	Misspelled.
0020 RANDOM (I)	No operands allowed.

9.43 READ Statement

1 READ <variable list>

2 READ #<channel>: <variable list>

3 READ #<channel>,<key>: <variable list>

Where:

```
<variable list> ::= <variable>[,<variable list>]
<channel> ::= <numeric expression>
<key> ::= <numeric expression>
          <string expression>
```

See also: DATA, INPUT, LINPUT, MAT INPUT, MAT READ, OPEN, READNEXT, and RESTORE statements

Purpose:

The READ statement is used to: accept data from DATA statements (format 1); accept data from a sequentially formatted file (format 2); accept data from an indexed or direct formatted file (format 3).

Comment:

The READ statement, format 1, causes the variables listed to be assigned values from the next data elements of the DATA statement. If there is more than one DATA statement in the program, then, when the first DATA statement's elements are used up, the next data element will come from the next DATA statement in the program. When there are no more DATA statements in the program, an "Out of data" error will occur when a READ is executed.

When it becomes necessary to use the same data more than once in a program, the RESTORE statement makes it possible to recycle through the complete set of DATA statements in the program or a partial set.

The other two formats of the READ statement operate similar to the INPUT statement discussed earlier. The primary difference between the READ statement and the INPUT statement (and LINPUT) is that the INPUT accepts ASCII data only (i.e., quoted strings and characters) and the READ statement accepts fields of data in internal BASIC format.

Formats 2 and 3 of the READ statement accept data from a file that was created with its complementary WRITE statement.

The READ statement can only access an I/O channel that was opened with access mode INPUT or UPDATE, not OUTPUT.

Format 2 of the READ statement accesses a file opened with SEQUENTIAL access method.

Format 3 of the READ statement accesses a file opened with DIRECT or INDEXED access method. A numeric key is used for a file opened with DIRECT access and a string key is used for a file opened with INDEXED access.

After a format 2 or 3 READ is performed the EOF function will indicate whether or not the read was successful. The EOF function will return a true value on a SEQUENTIAL access READ if the end of file was encountered; on an INDEXED access READ if the record with the specified key could not be found; on a DIRECT access READ if the record read was deleted or never written to.

On a DIRECT access READ the trappable error "Invalid key" will occur when an attempt is made to access a negative or zero record number or a record number greater than the maximum number of records in the file.

```
Examples:                                Explanation:

0010 READ A                              The value 1.23 is assigned to A.
  .
  .
0040 READ B,C                            The value 2.34 is assigned to B,
  .                                      the value 3.45 is assigned to C.
  .
0050 RESTORE 9010                        Next data element will come from
  .                                      line number 9010.
0100 READ A$                             The literal '1.23' is assigned to A$.
  .
  .
0130 RESTORE 8000                        Next data element will come from
  .                                      line number 9010.
  .
8000 . . .
9010 DATA 1.23, 2.34, 3.45,LITERAL,2ND LITERAL
9020 DATA 2.234, ABCDEF,ABCDE  FGHIJK,"  ABCDE FGHIJK  "

0010 OPEN #1: "DATA.FILE",INPUT DIRECT
0020 OPEN #2: "TEST.FILE",UPDATE SEQUENTIAL,EXTEND
0030 OPEN #3: "FILE.DATA",INPUT INDEXED
0040 READ #1,13: A$,B$,C,D$              The 13th record is read
0050 READ #2: B$,C$,A                    The next record is read
0060 READ #3,KEY$: FLD1$,FLD2$,TOTAL     Record with key matching contents of
                                         KEY$ is read.
```

```
Incorrect examples:                      Explanation:

0010 READ A                              First data element is alpha -
  .                                      'Conversion Error' will occur.
  .
0040 READ B                              No data elements left.
9000 DATA ABCD
```

9.44 READNEXT Statement

1 READNEXT #<channel>,<key>: <variable list>

Where:

<channel> ::= <numeric expression>
<key> ::= <string expression>
<variable list> ::= <variable>[,<variable list>]

See also: INPUT, LINPUT, MAT INPUT, MAT READ, and READ statements

Purpose:

The READNEXT statement will access the next record following the previous READ, WRITE, or READNEXT from an indexed file.

Comment:

This statement is very similar to the READ statement, however this statement only operates on a file opened with access method INDEXED. The key must be a string variable, not an expression.

When the READNEXT statement is executed the indexed disk file specified by the <channel> is read in a sequential manner. The record read by the READNEXT statement is the record whose key is the next key greater than the last record key accessed in this file. If there are no records whose key is greater than the last record accessed then the file pointer is considered to be at end-of-file and the EOF function may be used to detect this condition.

If a record is read by the READNEXT statement then the contents of that record's key is placed into the variable <key> and the contents of the individual fields of that record are placed into the variables specified in the <variable list>.

When an indexed file is first OPENed, the file pointer is positioned before the first record in the file. Therefore if the first access to an indexed file is a READNEXT statement then that statement will retrieve the first record in the file, if any exist. Each access of an indexed file by a READNEXT statement causes the file pointer to be advanced to the next record. Access to an indexed file by the READ statement causes the file pointer to be positioned to the record specified by that READ statement. (If the READ statement is unsuccessful the file pointer is positioned to the place that the record would have been at, if it had existed; therefore a READNEXT statement, following an unsuccessful READ statement, will retrieve the next record that logically follows the record searched for with the READ statement.)

An attempt to use the READNEXT statement to access a record created with a PRINT statement will cause an "Invalid file format" error. The READ and READNEXT statement can only access records created with the WRITE statement.

Examples: Explanation:

If a indexed file contains records with the following keys:

```
000100
000124
001001
003234
003235
004000
```

then the following statements will print the string "003234"

```
0100 READ #1,"002000":A$              Position after record 001001
0110 READNEXT #1,KEY$:A$              Get record following, i.e. 003234
0120 PRINT KEY$
```

9.45 REM Statement

```
1 REM
2 REM <unquoted string literal.
```

Purpose:

The REM statement allows the insertion of a comment or remark into a program.

Comment:

REM statements are valid BASIC statements and may be used anywhere that a statement can be used. They are saved as part of the program and appear whenever the program is listed, however they are ignored when the program is executed.

All characters after REM are ignored by the BASIC statement analyzer. For this reason, the REM statement must always be the last statement on a line.

The REM statement should never be used on the same line as a DATA statement. This is explained in the section on the DATA statement.

Examples:	Explanation:
0010 REMARK: THIS IS A REMARK 0020 REM: THIS IS A REMARK 0040 LET A = B \REM THIS IS A REMARK	Recommended syntax for using a REM on the same line as a statement.

Incorrect examples:	Explanation:
0010 DATA 1,2,3,4,5, \REM ABCDEF	The REM will be treated as a DATA element.
0020 LET A=B REM This is a remark	Statement separator missing.

9.46 RESTORE Statement

1 RESTORE

2 RESTORE <line number>

See also: READ and DATA statements

Purpose:

The RESTORE statement is used to re-use data elements from the DATA statements.

Comment:

When it is neccessary to use the same data elements from the DATA statements more than once in a program the RESTORE statement makes it possible to recyle through the complete set or a partial set of the DATA statements.

If the line number option is used the referenced line need not be of a DATA statement.

When the RESTORE statement is executed the internal pointer used for accessing the data elements of a program is set to point to the beginning of the program (line reference option not used) or to the line referenced. In either case the next READ statement will read the first data element at, or following, the statement pointed to.

Note: The interpretive mode of OASIS BASIC does not allow the RESTORE statement to be used with a line label reference. However, since the compiler translates all line label references to statement address references a program using the RESTORE statement with a line label reference would execute using the compiled version of the program.

Examples:	Explanation:
0050 RESTORE	The next READ will read the first data element of the first DATA statement in the program.
0060 RESTORE 1	Same as line 50.
0070 RESTORE 9000	The next READ will read the first data element of the first DATA statement at or following line 9000.
0080 RESTORE 9900	The next READ will read the first data element of the first DATA statement at or following the line number 9900.

9.47 RESUME Statement

1 RESUME

2 RESUME 0

3 RESUME <line reference>

Where:

<line reference> ::= <line number>
<line label>

See also: ON ERROR statement

Purpose:

The RESUME statement terminates an error handling routine and specifies what to do next.

Comment:

The RESUME statement acts like a RETURN statement except that it may only be used in an error handling routine.

After an error handling routine has performed the tasks required for the specific error (see ON ERROR statement) the routine must return control to BASIC. The RESUME statement performs this task. The RESUME statement must be used to return control from an error handling routine. If the error routine does not use the RESUME statement then BASIC will continue executing the program but no errors will be trapped (the program becomes a "large" error routine).

At this time BASIC needs to know what was done and what to do. There are three possible situations that might exist: 1) the error was corrected by the error routine and the statement that caused the error is to be re-executed; 2) the error could not be corrected by the routine and the system is to handle the error; 3) the error was corrected by the routine but a different statement is to be executed.

These three situations correspond to the three formats of the RESUME statement:

RESUME with no line reference (format 1) indicates that BASIC is to ignore the error and to re-execute the statement causing the error.

RESUME 0 (format 2) indicates that BASIC is to handle the error. In this event BASIC will display the error message corresponding to the error along with the line number of the statement causing the error (ERL). If the program was executed from the RUN environment then BASIC will be exited; if the program was executed from the BASIC environment then the command mode of BASIC will be entered (prompt character of "-").

RESUME <line reference> (format 3) indicates that the error was corrected but control is to be transferred to the line specified.

Examples:	Explanation:
9000 IF ERR=2 THEN 9020 9005 IF ERR=1 THEN 9030	
9010 RESUME 0	Error cannot be handled - this lets BASIC handle it.
9020 RESUME	Error was corrected (or ignored) and the program resumes execution at the statement causing the error.
9030 RESUME EXIT	Error was corrected (or ignored) and control is to be transferred to the line with the label EXIT.

9.48 RETURN Statement

1 RETURN

2 RETURN <line reference>

Where:

<line reference> ::= <line number>
<line label>

See also GOSUB and ON GOSUB statements

Purpose:

The RETURN statements terminates the execution of a subroutine and transfers control back to the statement following the call (GOSUB) to the subroutine.

Comment:

There may be more than one RETURN statement in a subroutine, however, the first one executed causes the subroutine to terminate. It is a good programming practice to have only one RETURN statement in a subroutine and, if multiple exit points are needed, branch to that one statement from the various parts of the subroutine. This makes the routine easier to read and maintain.

The RETURN statement cannot be executed without a previous execution of a GOSUB statement. When this is attempted a "Return stack empty" error occurs.

When a line is referenced on the RETURN statement the referenced line must exist in the program (same as the GOTO statement).

The RETURN statement with the optional line number reference used causes the location of the statement following the GOSUB call to be discarded and control transfers to the line referenced.

It is bad practice to use the line reference option except in unusual or exceptional cases. A better, and approved method of performing a similar function, is to use the SELECT or WHILE statement structures.

Examples:	Explanation:
0010 GOSUB 30	Execute subroutine at line 30
0020 PRINT A$ GOTO 9000	Statements executed after RETURN
0030 REM Subroutine entry	Beginning of subroutine
.	
.	
0090 RETURN	Exit subroutine
0100 GOSUB INPUT	Execute subroutine a label INPUT
.	
.	
0500 INPUT: REM Input subroutine	Beginning of subroutine
.	
.	
0590 RETURN CLOSE.UP	Exit subroutine and transfer control to CLOSE.UP label.

9.49 RUN Statement

1 RUN

2 RUN <program name>

Where:

```
<program name> ::= <file name>[.<file type>][:<file disk>]
<file type> ::= BASICOBJ (with BASIC)
                BASICCOM (with RUN)
```

See also: CHAIN, CLEAR and LINK statements

Purpose:

The RUN statement allows the user to execute a program already in memory or one stored on disk.

Comment:

When <program name> is not specified, the program currently in memory is executed, starting with the first line of the program.

Before the RUN statement is executed, a CLEAR command is automatically executed.

<program name>, when specified, must be a string expression. When BASIC is being used (not the compiler RUN time command) only BASICOBJ files will be searched for. When RUN is being used (not the interactive interpreter) only BASICCOM files will be searched for.

When the <program name> is specified, a search is made for the program. If the program is found, a NEW command is executed and the specified program is loaded. Execution begins with the smallest line number.

Previous versions of OASIS BASIC supported the <line number> operand. The recommended method of transferring control to another program at a specific line is the use of a control variable (defined as COMMON) that is tested by an ON-GOTO statement at the start of the program transferred to.

The CHAIN, RUN, and LINK statements all perform similar tasks, but with significant differences:

Program Linkage Statements

Statement	I/O Channels	Variables	COMMON
RUN	Closed	Cleared	Cleared
CHAIN	Closed	Cleared	Not cleared
LINK	Not closed	Cleared	Not cleared

Examples:	Explanation:
-LOAD TEST	Program "TEST" is loaded,
-RUN	then executed.
-RUN TEST	Same as above.
1000 RUN	Re-execute program in memory.
1010 RUN "JOE"	Execute program named "JOE".

Incorrect examples	Explanation
10 RUN PROGRAM	Program name must be an expression.
20 RUN "PROGRAM" LABEL	Line labels not allowed.

9.50 SELECT Statement

1 SELECT

2 SELECT <expression>

See also: CASE, CEND and OTHERWISE statements

Purpose:

The SELECT statement defines the start of a SELECT-CASE-CEND programming structure.

Comment:

Format 1 of the SELECT statement specifies that subsequent, matching CASE statements will specify the complete relational expression that must evaluate true for the statements following to be executed.

Format 2 of the SELECT statement specifies the expression that is to be compared with the expression of subsequent, matching CASE statements.

SELECT structures may be nested to any depth.

The SELECT-CASE-CEND programming structure is a powerful aid to the programmer wishing to write structured programs in BASIC, a language that doesn't lend itself to structured programming techniques. (Also see ON ERROR, FOR-NEXT, IF-IFEND, and WHILE-WEND structures.)

Examples:	Explanation:
3000 SELECT RADIUS*2.*PI	Define VALUE
3010 CASE 0	
3020 SELECT	Perform only if VALUE=0
3030 CASE SUBVALUE%=20	
3040 .	Perform only if VALUE=0 and SUBVALUE%=20
3050 CASE SUBVALUE%>32	
3060 .	Perform only if VALUE=0 and SUBVALUE%>32
3062 CASE ERROR%	
3064 .	Perform only if VALUE=0 and ERROR%<>0
3070 CEND	End of nested SELECT structure
3080 CASE I-14	
3090 .	Perform only if VALUE=I-14
3110 CASE J	
3120 .	Perform only if VALUE=J%
3130 CEND	End of SELECT structure

9.51 SLEEP Statement

1 SLEEP <integer expression>

Purpose:

The SLEEP statement causes BASIC to pause for a period of time, allowing the operator time to read a message, etc.

Comment:

The value of <integer expression> is rounded up and integerized. The value of this expression must be between 0 and 32767 (approximately 9 hours), inclusive.

The minimum time that the SLEEP statement will pause is one second. Specifying any value less than one will be interpreted as the default, one second.

Examples	Explanation
10 SLEEP 10	Suspend processing for 10 seconds.
95 SLEEP X/4	Wait for one fourth of X value.
200 SLEEP .5	Wait for one second.

9.52 STOP Statement

1 STOP

2 STOP <expression>

See also: END and QUIT statements

Purpose:

The STOP statement terminates execution of a program without closing any files nor altering working storage.

Comment:

The STOP, END and QUIT statements all terminate execution of a program. The QUIT and END statements are the normal termination of a program in a non-development mode.

The STOP statement is used when an abnormal exit from the program is desired, as needed during the development and debugging of a program. When it is executed, the status of the program remains unchanged, and the message "STOP at Line nnnn" is displayed on the terminal. BASIC will enter the command mode (prompt character of "_").

If a STOP statement was executed, a CONTINUE command will resume execution at the statement following the STOP statement. This allows the programmer to examine or alter portions of the program or to change the value of some variables.

When an expression is specified after the STOP verb that expression will be evaluated and displayed with the stop message: "STOP <value of expression> at line XXXX". This allows the programmer to put identifying messages on the screen to assist in the debugging.

Examples:	Explanation:
0010 STOP	Program stops execution and allows maintenance.
0020 STOP A$	Program stops execution, as above and displays the current value of the string A$.

9.53 THEN Statement

1 THEN [<statement>]

2 THEN [<line number>]

See also: ELSE and IF statements

Purpose:

The THEN statement specifies the action to be taken when a multiline IF statement relation is true.

Comment:

The THEN statement is only a statement when used in conjunction with the multi-line format of the IF statement. When used in this manner the verb THEN is optional.

<statement> may be any statement or statements, including another IF statement.

Format 2 of the THEN statement is an implied THEN GOTO <line number> statement.

Examples:

```
0010 IF A                                  Test A for non zero
0020     THEN GOSUB 2000                   Perform if A<>0
0030          PRINT USING "###",A                 "    "   "
0040          GOTO TOP.OF.PAGE                    "    "   "
0050     IFEND                             End of conditional execution

0010 IF VALUE > CONTROL                    Test expression
0020    THEN IF VALUE > LIMIT              Perform if expr is true
0030       THEN GOSUB ERROR                Perform if both expr are true
0040            GOTO EXIT                        "    "   "    "   "
0050       ELSE IF ERR.NUM < ERR.LIMIT THEN QUIT
                                           Perform only if first expr is true
                                           and second expr is false
0060       IFEND                           End conditional execution from second expr
0070    IFEND                              End of conditional execution
```

Incorrect Examples:

```
0010 IF VALUE>5 THEN 100
0020      THEN PRINT "XYZ"                 Not in a multi-line IF statement
```

9.54 UNLOCK Statement

1 UNLOCK #<channel>

Where:

<channel> ::= <numeric expression>

See also: CHAIN, CLOSE, DELETE, INPUT, LINPUT, MAT INPUT, MAT PRINT, MAT READ, MAT WRITE, OPEN, PRINT, PRINT USING, READ, READNEXT, and WRITE statements

Purpose:

The UNLOCK statement operates in multi-user OASIS only and allows a program to release a record for other users use.

Comment:

The UNLOCK statement is only effective when the channel was opened with UPDATE access, not INPUT or OUTPUT.

The UNLOCK statement releases the record read from the channel with an INPUT, MAT INPUT, MAT READ, READ, or READNEXT statement. After the UNLOCK statement is executed another user partition may read the record just released.

An unlock function is performed automatically when any of the following statements is executed: CLOSE, DELETE, INPUT, LINPUT, MAT INPUT, MAT PRINT, MAT READ, MAT WRITE, PRINT, PRINT USING, READ, READNEXT, and WRITE. Note that the input type statements may lock another record.

Examples:	Explanation:
0010 OPEN #1: "DATA.FILE",UPDATE SEQUENTIAL	
0020 READ #1: RECORD$	Read the first record and locks it.
0030 UNLOCK #1	Releases the record for others use.

9.55 WAIT Statement

```
1 WAIT

2 WAIT DEVICE <device number>

3 WAIT MEMORY <address>,<and mask>[,<xor mask>]

4 WAIT PORT <port>,<and mask>[,<xor mask>]

Where:

     <device number> ::= <numeric expression>
     <address> ::= <numeric expression>
     <port> ::= <numeric expression>
     <and mask> ::= <numeric expression>
     <xor mask> ::= <numeric expression>

See also: GET and PUT statements
```

Purpose:

The WAIT statement suspends execution until some event has occurred.

Comment:

The most frequent use of this statement (format 1) is to suspend operation until the operator has typed any key on the console keyboard. This use is the same as the systems when it displays a page of information and then waits for the operator to release that page before displaying the next page. An up-arrow character (^) will be displayed in the bottom, left hand corner of the screen while the system is waiting for the operators response. This is a conditional wait determined by the status of the System Screen-wait key (see OASIS System Reference Manual).

This statement causes BASIC to test a byte from the specified device (format 2), memory address (format 3), or port (format 4), logically AND it with <and mask> and logically eXclusive OR it with <xor mask>. The statement is re-executed if the result is not zero (true).

The <port> expression must evaluate to an integer between 0 and 255; the <device number> expression must evaluate to an integer between 9 and 32; the <address> expression must evaluate to an integer in the range -32767 - +32767. (This value, unlike other integers, is interpreted as an unsigned value, which automatically adjusts its range to 0 - 65535.)

If <xor mask> is omitted, it is assumed to be equal to zero.

This statement can be very useful for waiting for an I/O device to become ready for output, or waiting for a character to be input from a device.

The WAIT DEVICE has no masks available because it returns control to BASIC as soon as any change (non zero) occurs with the device.

The WAIT statement does not read the data from the port or device, only the status of the device or port is tested. This statement would normally be used to determine the time that an event happened in order to synchronize two processes.

Examples:	Explanation:
0010 WAIT DEVICE 9	The program suspends execution until a key is entered from the console keyboard. When any key is typed the program will continue execution with the statement following.
0020 WAIT PORT 25,0FH	The program suspends execution until a byte is input on port 25 that has the four low-order bits off.
0030 WAIT	Wait for operator to release current page of data on screen.

9.56 WEND Statement

1 WEND

See also: WHILE statement

Purpose:

The WEND statement marks the end of a WHILE-WEND programming structure.

Comment:

The WEND statement requires that a corresponding WHILE statement exists and that the WHILE statement must have been executed prior to the WEND statement.

The WEND statement performs two functions: marks the end of a WHILE-WEND structure--the statement following the WEND statement is executed when the expression in the WHILE statement is false; causes the corresponding WHILE statement to be re-executed when the expression of that WHILE statement was true the last time.

WHILE-WEND structures may be nested to any depth.

Example:	Explanation:
0010 WHILE CONTROL%	Test the variable CONTROL%
0020 GOSUB 1000	Perform if CONTROL% is non-zero
0030 GOSUB 1200	" " "
0040 WHILE OPTION$="HELP"	" " "
0050 GOSUB HELP.ROUTINE	Perform if CONTROL%<>0 AND OPTION$="HELP"
0060 OPTION$=""	" " " " " "
0070 WEND	
0080 WEND	Go back to 10 if CONTROL% was non-zero

9.57 WHILE Statement

1 WHILE <numeric expression>

2 WHILE <logical expression>

3 WHILE <relational expression>

See also: WEND statement

Purpose:

The WHILE statement marks the beginning and qualifying condition of a WHILE-WEND programming structure.

Comment:

The WHILE statement requires a corresponding WEND statement, which marks the end of the WHILE-WEND structure.

When the WHILE statement is encountered the expression is evaluated. If the result of the expression is non-zero or true the statements following are executed. If the result of the expression is zero or false then the statements following, up to and including, the corresponding WEND statement are skipped.

If the expression was true and the statements were executed, when BASIC encounters the corresponding WEND statement control will be transferred back to this WHILE statement for expression re-evaluation. Because of this looping feature, there should be some statement within the loop that could modify the results of the expression evaluation, or a statement that will transfer control out of the loop; otherwise the loop will be executed indefinitely.

WHILE-WEND structures may be nested to any depth.

Example:

```
0010 WHILE A%<10
0020     A% = A%+I%
0030     FOR I%=1 TO 5
0040          PRINT I%
0050          NEXT I%
0060     WEND
0070 PRINT A%
```

Explanation:

```
Test the expression
Perform only if true.
   "      "       "
   "      "       "
   "      "       "
If exp was true then go back to 10
Otherwise perform this and continue.
```

```
0010 IF NOT (A%<10) THEN 70
0020      A% = A%+I%
0030      FOR I%=1 TO 5
0040           PRINT I%
0050           NEXT
0060      GOTO 10
0070 PRINT A%
```

This is the same as above example.

9.58 WRITE Statement

1 WRITE #<channel>: <expression list>

2 WRITE #<channel>,<key>: <expression list>

Where:

```
<channel> ::= <numeric expression>
<expression list> ::= <expression>[,<expression list>]
<key> ::= <numeric expression>
          <string expression>
```

See also: DELETE, MAT PRINT, MAT WRITE, PRINT, and PRINT USING statements

Purpose:

The WRITE statement allows the user to create or update sequential, direct or indexed file records.

Comment:

<channel> is the internal I/O channel number of a channel that was opened for OUTPUT or UPDATE that does not have write protect status. If an attempt is made to write to a protected file, the error message "Protected File" will be displayed.

Format 1 of the WRITE statement is used for sequential format files opened with access method of SEQUENTIAL. This format causes the next record in sequence to be written to the file (i.e. if the last record written to the file was the 11th record then this statement will write the 12th record to the file).

Format 2 of the WRITE statement is used for files opened with access method of DIRECT or INDEXED. A file opened with access method DIRECT will require a numeric key expression; a file opened with access method INDEXED will require a string key expression. In either case the record specified by the key will be written to the file, replacing any existing record with the same key.

When the key is numeric its value must be greater than zero and less than or equal to the number of records allocated to the file. Using a key outside of this range will cause an "Invalid key" error.

The WRITE statement always locks the record before writing it to the file. The WRITE statement also unlocks any record that was locked in the file by this program (unless option LOCK was used with the OPEN statement).

The only proper way to retrieve a record written to a disk file with the WRITE statement is with a READ or READNEXT statement. Using an INPUT or LINPUT statement on a record that was output with a WRITE statement will cause an "Invalid file format" error.

Examples:

```
0010 OPEN #1: "DATA.FILE",OUTPUT SEQUENTIAL
0020 OPEN #2: "CUSTOMER.MASTER",UPDATE INDEXED
0030 OPEN #3: "TRANSACT.DETAIL:A",UPDATE DIRECT,LOCK
0040 WRITE #1: DATA1,DATA2,STRING$,1*34+5
0050 WRITE #2,"Name": ADDR$,CITY$,STATE$,FORMAT$(ZIP,"99999"),BALANCE
0060 WRITE #3,24: A,B,C,D,E,F,TOTAL,LINK%
```

Incorrect Examples:	Explanation:
0070 WRITE #1,23: A,B,C,D	Not valid for sequential access.
0080 WRITE #2: A$,BETA$,C	Indexed access requires key.
0090 WRITE #3,"REC"&STR(I%): A,B	Direct access requires numeric key.

(This page intentionally left blank)

CHAPTER 10

FUNCTIONS

A function is a relation between two variables such that for each value of the independent variable there is one, and only one, value of the dependent variable. When a function is used (called) in BASIC, the independent variable(s) is the parameter and the dependent variable is the value of the function. For example:

```
100 LET Y = SQR(X)
```

X is the independent variable and must be defined before the function, SQR, is called. The value of the function, SQR(X) is the dependent variable and, in this example, is assigned to the variable Y.

Functions are not statements.

BASIC provides many predefined functions for the programmer's use. These include thirty numeric functions (including trigonometric), twenty six string functions, four input/output functions, one file function, four logical functions, two error functions, and one user function. Specifically they are:

Numeric functions:

ABS	Absolute value	MAX	Return maximum of two numbers
ASC	Decimal value of character	MIN	Return minimum of two numbers
ATN	Arctangent	MOD	Perform modulo of number
BIN	Convert from binary base	NBR	Test string for numerics
COS	Cosine	OCT	Convert from octal base
DAY	Convert from ext date format	PI	Constant: 3.141592653590
EXP	Exponential	RND	Pseudorandom number
FIX	Integerize number	ROUND	Round number
FLOAT	Float integer number	SCH	Search string for sub-string
HEX	Convert from hexadecimal base	SEC	Convert from ext time format
INT	Return integer portion	SGN	Return sign of value
LEN	Return length of string	SIN	Sine
LOG	Natural logarithm	SQR	Square root of number
MATCH	Compare string with mask	TAN	Tangent
		VAL	Numeric value of string number

String functions:

AT$	Cursor control	LTRIM$	Remove leading spaces
BINOF$	Convert to binary base	MID$	Return middle of string
CHR$	Return ASCII of number	OCTOF$	Convert to octal base
CRT$	Cursor control	OVR$	Overlay string with string
DATE$	Convert to ext date	REP$	Replace sub-string field
DEL$	Delete sub-string field	RIGHT$	Return right portion of string
DTE$	Validate string for date	RPAD$	Add trailing spaces
EXT$	Extract sub-string field	RPT$	Generate string of characters
FORMAT$	Format string	RTRIM$	Remove trailing spaces
HEXOF$	Convert to hexadecimal base	SPACE$	Generate string of spaces
INS$	Insert sub-string field	STR$	Return ASCII value of character
LEFT$	Return left portion of string	TIME$	Convert to ext time format
LPAD$	Add leading spaces	TRIM$	Remove leading & trailing spaces

Input/Output functions:

INP	Value of control char entered	PAGE	Return page length of channel
LINE	Return line length of channel	POS	Position of output rec pointer

Logical functions

LRL	Logical rotate left	LSL	Logical shift left
LRR	Logical rotate right	LSR	Logical shift right

File function:

EOF	Test for end of file

Error functions:

ERL	Line number of error	ERR	Error number of error

User function:

USR	User written assembly language subroutine call.

The following functions always return an integer value: ASC, EOF, ERL, ERR, FIX, HEX, INP, LEN, LINE, LRL, LRR, LSL, LSR, MATCH, NBR, PAGE, POS, SCH, SGN.

The following functions return an integer value when the parameter to the function is an integer: INT and USR.

All other numeric functions return floating point values.

A function call has the general form of:

<function name>[$](parameters)

In addition to the pre-defined functions listed above, the user may define his own functions with the DEF statement. These functions are only defined while the program defining them is in memory.

The parameters passed to a function are not changed by the function.

Function names cannot be abbreviated and function names cannot be used as variable names.

References to string functions do not require the dollar sign character. For example, SPACE(5) is acceptable for SPACE$(5).

10.1 Numeric Functions

ABS(<num-exp>) The numeric expression is evaluated and its absolute value is assigned to the function.

```
Example:  PRINT ABS(23);ABS(-23)
           23  23
```

ASC(<string-exp>) The string expression is evaluated and the ASCII, integer value of the first character in the resulting string is returned.

```
Example:  PRINT ASC(A$)
           65
```

BIN(<string-exp>) The string expression is evaluated and the resulting string is interpreted as a binary value with its equivalent decimal, integer value returned. Remember that binary values only use the digits 0 and 1.

```
Example:  PRINT BIN("0101010101010101");BIN("0000111100001111")
           21845  3855
```

DAY(<string-exp>) The string expression is evaluated and interpreted as a date field according to the currently set DATEFORM (DATEFORM 3 is interpreted as DATEFORM 2). Non numeric characters in the string expression are interpreted as delimiters between the month, day, and year. The number of days since December 31,1899 to that date is returned. An invalid date string expression will cause the function to return a -1.

```
Example:  PRINT DAY("5/17/77"),DAY("1-1-0")
           28261          1
```

EXP(<num-exp>) The expression is evaluated; the constant e is raised to the value of the expresion and assigned to the function.

FIX(<num-exp>) The fractional portion of the value of the expression is truncated; the resulting integer portion is assigned to the function (32767 to -32767).

```
Example:  PRINT FIX(1.5);FIX(.5);FIX(5.5);FIX(-43.5)
           1  0  5 -43
```

FLOAT(<num-exp>) The numeric expression is evaluated and converted, if necessary, to a floating point value.

```
Example:  PRINT 1/4;1/FLOAT(4);1/4.
           0  .25  .25
```

HEX(<string-exp>) The string expression is evaluated and the resulting string is intepreted as a hexadecimal value with its equivalent decimal, integer value returned. Remember that hexadecimal values use the digits 0 through 9 and the letters A through F.

```
Example:  PRINT HEX("0FF");HEX("100")
           255  256
```

INT(<num-exp>) The expression is evaluated and the greatest signed integer of that value is assigned to the function. The result of this function is an integer or floating point, depending upon the argument of the function.

```
Example:  PRINT INT(1.5);INT(.5);INT(-4.6)
           1  0 -5
```

LEN(<string-exp>) The string expression is evaluated and its length is returned as an integer.

```
Example:  PRINT LEN("ABCDEF");LEN("     X     ");LEN(SPACE$(10))
           6  10  10
```

LOG(<num-exp>) The expression is evaluated and the natural logarithm of that value is assigned to the function. (Natural logarithms are logarithms to base e).

The common logarithm (base 10) may be computed by dividing the

natural logarithm by LOG(10), i.e.: LOG10(X) = LOG(X)/LOG(10).

MATCH(<string-exp1>,<string-exp2>) The two string expressions are evaluated and the second expression is used as a mask for match purposes. If the first string does match the mask a true value is returned (-1); if the string does not match the mask a false value is returned (0). The mask characters are interpreted as follows:

- @ Any alphabetic character or space in this position is a match.
- # Any numeric digit in this position is a match.
- ? Any character in this position is a match.
- *@ One or more alphabetic characters in these positions will match.
- *# One or more numeric digits in these position will match.
- *? One or more characters in these positions will match.
- % This is the 'escape' character: the special character following (@, #, ?, * or %) is treated as a literal match character).

All other characters are treated as literal match characters, i.e., the corresponding position in the first string must contain the specific character.

The following are example masks along with a description of what they will match:

Mask: "ABC?"
Matches: Any four character string starting with the uppercase letters A, B, and C. The following strings will match this mask: "ABCX", "ABC1", "ABC-". The following strings will not match this mask: "ABDE", "XXXX", "ABCDEFGH", "ABC", "WXYZ".

Mask: "ABC*?"
Matches: Any four or more character string starting with the uppercase letters A, B, and C. The following strings will match this mask: "ABCDEF", "ABC#*$$234". The following strings will not match this mask: "A", "234", "ABDXLKJ".

Mask: "ABC*?DEF"
Matches: Any string whose first three letters are A, B, and C, and whose last three letters are D, E, and F. One or more characters between these are acceptable. The following strings will match this mask: "ABCXDEF", "ABCXXXXXDEF", "ABC12433ABCDEF". The following strings will not match this mask: "ABC", "ABCD", "ABCDE", "ADEF", "ABCDEF".

Mask: "###-##-####"
Matches: Any eleven character string with: three digits, a hyphen, two digits, a hyphen, and four digits (like a Social Security Number). The following string will match this mask: "123-45-6789". The following strings will not match this mask: "123456789", "123/45/6789", "12ABD".

Mask: "@@@###"
Matches: Any six character string whose first three characters are letters or spaces and whose last three characters are digits. The following strings will match this mask: "abc123", "AB 123", "Xyz002". The following strings will not match this mask: "123ABC", "AB1234", "XXXXXX", "ABCX123".

Mask: "%*%*%*###"
Matches: Any six character string whose first three characters are asterisks and whose last three characters are digits. The following strings will match this mask: "***123", "***738". The following strings will not match this mask: "***ABC", "***1234", "112456", "**ABCDEFG".

MAX(<num-exp1>,<num-exp2>) The two expressions are evaluated and compared to each other. The value of the expression whose value is greatest is

returned.

Example: PRINT MAX(5,21);MAX(PI,3.14);MAX(1,1)
21 3.141592653590 1

MIN(<num-exp1>,<num-exp2>) The two expressions are evaluated and compared to each other. The value of the expression whose value is smallest is returned.

Example: PRINT MIN(5,21);MIN(1,-1);MIN(3*23,70)
5 -1 69

MOD(<num-exp1>,<num-exp2>) The two numeric expressions are evaluated. The value of the first expression is divided by the value of the second expression and the remainder is assigned to the function.

Example: PRINT MOD(11,4);MOD(2.2,.8)
3 .6

NBR(<string-exp>) Analyzes the string expression to determine if it could be converted to a number. The string expression is first evaluated. If the resulting string contains any non-numeric characters (other than digits, plus or minus sign, period (or comma if OPTION COMMA is in effect), leading or trailing spaces, or letter E) an integer 0 is returned (false). If the resulting string is a valid decimal or hexadecimal number then an integer -1 is returned (true).

Example: PRINT NBR("123");NBR("0ABCH");NBR("1.23E23")
-1 0 -1

PRINT NBR("NAME")
0

OCT(<string-exp>) The string expression is evaluated and the resulting string is interpreted as a octal value with its equivalent decimal, integer value returned. Remember that octal values only use the digits 0 through 7.

Example: PRINT OCT("071");OCT("100")
57 64

PI The constant 3.141592653590 is assigned to the function.

RND The value of the next pseudorandom number is assigned to the function. The value is a floating point number between zero and one.

ROUND(<num-exp1>,<num-exp2>) The two numeric expressions are evaluated and the first expression is rounded to the number of places specified by the value of the second expression. Positive values for the second expression indicate the number of digits to the right of the decimal point; negative values for the second expression indicate the number of digits to the left of the decimal point.

Example: PRINT ROUND(PI,4);ROUND(1234.567,-2)
3.1416 1200

PRINT ROUND(1.234567,4);ROUND(2.34,0)
1.2346 2

SCH(<num-exp>,<string-exp1>,<string-exp2>) The expressions are all evaluated. A search is made of the resulting <string one>, starting at the character position <number one>, for the sub-string <string two>.

If <string two> is found in <string one> then the starting position in <string one> is returned. If <string two> is not found in <string one> then the integer value zero is returned (false).

When <string two> is the null string (equal to "") the integer value one is always returned. The null string is a proper substring of any string and is treated conventionally as the first element of every string.

```
Example:  PRINT SCH(1,"ABCDEFGH","D");SCH(3,"ABCDEFGH","EFG")
           4  5

          PRINT SCH(1,"ABCDEFGH","X");SCH(1,"ABC","")
           0  1
```

SEC(<string-exp>) The string expression is evaluated and interpreted as a normalized time of day (hh:mm:ss). The value of the number of seconds since midnight (00:00:00) to the time represented by the string expression is returned. The times input to this function may use any non-numeric character for delimiters between hours, minutes, and seconds (except semicolon and comma). For example, valid input to this function includes: "1.1.1", "1-1+1", "1H1M1S", "1 1", "1", etc.

```
Example:  PRINT SEC("12:00:00");SEC("01:05:08");SEC("2.3")
           43200  3908  7380
```

Note: To get the current time of day in seconds use: SEC(TIME$(0))

SGN(<num-exp>) The numeric expression is evaluated and the sign (+1, 0 or -1) of the value is assigned to the function.

```
Example:  PRINT SGN(PI);SGN(-1.0/-2.0);SGN(-43);SGN(PI-PI)
           1  1 -1  0
```

SQR(<num-exp>) The expression is evaluated and the square root of the resulting value is assigned to the function.

```
Example:  PRINT SQR(4);SQR(25);SQR(11)
           2  5  3.31662479161...
```

VAL(<string-exp>) The string expression is evaluated and interpreted as a numeric constant. If the string contains any non-numeric characters (see section on "Numeric Constants") a trappable error occurs. If the string is a valid number then the value of that number is assigned to the function.

```
Example:  PRINT VAL("123");VAL("1.234E23")
           123  1.234E+023

          PRINT VAL("ABCD")
          Illegal number
```

10.2 Trigonometric Functions

ATN(<exp>) The expression is evaluated and the arctangent of that value is assigned to the function.

COS(<exp>) The expression is evaluated and the cosine of that value is assigned to the function.

SIN(<exp>) The expression is evaluated and the sine of that value is assigned to the function.

TAN(<exp>) The expression is evaluated and the tangent of that value is assigned to the function.

The argument for the SINe, COSine, and TANgent functions is an angle expressed in radians. Although any angle will be accepted as a valid argument, some accuracy will be lost if the angle is outside the range of plus or minus 2PI. This is because the function routine must first reduce the angle to the first quadrant before evaluating the function. If the angle is known in degrees, it must be converted to radians before it is used as the function argument. This may be done as part of the expression.

The argument of the ArcTaNgent function may be any number (the tangent of any angle). The result will be an angle in the range plus or minus PI/2 radians.

The following identities may be used to compute trigonometric functions other than sine, cosine, tangent, and arctangent:

Function	Identity
Cotangent	DEF FNCOT(ANGLE) = 1/TAN(ANGLE)
Secant	DEF FNSEC(ANGLE) = 1/COS(ANGLE)
Cosecant	DEF FNCOSEC(ANGLE) = 1/SIN(ANGLE)
Arcsine	DEF FNARCSIN(ANGLE) = ATN(ANGLE/SQR(1-ANGLE^2))
Arccosine	DEF FNARCCOS(ANGLE) = ATN(SQR(1-ANGLE^2)/ANGLE)
Arccotangent	DEF FNARCCOTAN(ANGLE) = ATN(1/ANGLE)
Arcsecant	DEF FNARCSEC(ANGLE) = ATN(SQR(ANGLE^2-1))
Arccosecant	DEF FNARCCOSEC(ANGLE) = ATN(1/SQR(ANGLE^2-1))
Degrees to Radians	DEF FNRAD(ANGLE) = ANGLE*PI/180
Radians to Degrees	DEF FNDEG(ANGLE) = ANGLE*180/PI

Hyperbolic Function	Identity
Hyperbolic sine	DEF FNHSIN(ANGLE) = (EXP(ANGLE)-EXP(-ANGLE))/2
Hyperbolic cosine	DEF FNHCOS(ANGLE) = (EXP(ANGLE)+EXP(-ANGLE))/2
Hyperbolic tangent	DEF FNHTAN(A) = (EXP(A)-EXP(-A))/(EXP(A)+EXP(-A))
Hyperbolic secant	DEF FNHSEC(ANGLE) = 1/FNHCOS(ANGLE)
Hyperbolic cosecant	DEF FNHCOSEC(ANGLE) = 1/FNHSIN(ANGLE)
Hyperbolic cotangent	DEF FNHCOTAN(ANGLE) = 1/FNHSEC(ANGLE)

10.3 String Functions

In the examples, assume that A$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

AT$(<num-exp1>,<num-exp2>) Generates a string of characters representing the cursor control commands for the terminal designated by the console terinal attachment. The first expression is interpreted as the horizontal coordinate. The second expression is interpreted as the vertical coordinate. Both coordinates are relative to one. For example, the upper left corner of the screen is referenced as AT$(1,1). Only values that are within the range of the attached console may be used. Any values greater than the maximum or less than one will cause the function call to be ignored.

BINOF$(<num-exp>) The numeric expression is evaluated, integerized and translated into the string of characters representing the value in binary. A sixteen character string is always generated.

```
Example:  PRINT BINOF$(123);" ";BINOF$(23129)
          0000000001111101 0101101001011001
```

CHR$(<num-exp>) Generates a one character string whose ASCII value is the value of the expression (see appendix on "Character Codes").

```
Example:  PRINT CHR$(65)
          A
```

CRT$(<num-exp1>,<num-exp2>) This is a synonym for the AT$ function (see above).

CRT$(<string-exp>) Generates a string of characters representing the cursor control commands for the terminal designated by the CONO attachment. Correct values for string expression and their functions are:

HOME	Move cursor to upper left corner.
CLEAR	Clear screen.
EOS	Erase to end of screen.
EOL	Erase to end of line.
UP	Move cursor up one line.
DOWN	Move cursor down one line.
LEFT	Move cursor one position to left.
RIGHT	Move cursor one position to right.
BELL	Sound the buzzer or bell on the console.
IL	Insert line.
DL	Delete line.
IC	Insert character.
DC	Delete character.
PON	Following characters are to be screen protected.
POFF	Following characters are not screen protected.
EU	Erase unprotected.
KON	Keyboard unlock.
KOFF	Keyboard lock.
FON	Format on.
FOFF	Format off.
BON	Following characters are to "blink".
BOFF	Following characters are normal (no blink).
ULON	Following characters are to be underlined.
ULOFF	Following characters are not to be underlined.
RVON	Following characters are to be displayed in reverse video (black on white background).
RVOFF	Following characters are to be displayed in normal video (white on black background).

Note: The control codes generated by this function are the internal codes used to perform the function. The code is only translated to the proper character sequence when it is output by the system.

This function always generates the internal code but it is only meaningful when that code is output to the CONSOLE. Refer to the OASIS System Reference Manual appendix on "Terminal Class Codes" for the specific controls implemented for each type of terminal class.

DATE$(<num-exp>) Returns a string of characters in normalized date format according to the currently set DATEFORM (DATEFORM 3 is interpreted the same sa DATEFORM 2) representing the expression interpreted as the number of days since December 31, 1899. The value 0 (zero) is interpreted as the current system date.

```
Example:  PRINT DATE$(10);" ";DATE$(0);" ";DATE$(28262)
          01/10/00 05/15/78 05/18/77
```

DEL$(<string-exp>,<num-exp1>,<num-exp2>) Returns the string expression with the subfield whose position in string is indicated by the values of the two numeric expressions deleted. The string deleted is the subfield of the string whose position is the <num-exp2> subfield of <num-exp1> subfield, including its delimiter.

```
Example:  B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

          PRINT DEL$(B$,2,0)
          AAAA^C1C1C1]C2C3C3]C3C3C3^DDDD

          PRINT DEL$(B$,3,3)
          AAAA^BBBB^C1C1C1]C2C2C2^DDDD

          PRINT DEL$(B$,3,0)
          AAAA^BBBB^DDDD
```

As illustrated, when the second numeric expression is zero the entire field referenced by the first numeric expression is deleted. When the field designated by two numeric expressions does not exist in the string, the string expresion is returned unmodified.

Note: Field and subfield delimiters are not really the characters ^ and] because the parity bit is turned on to indicate that the character is a delimiter and not a normal ASCII character.

DTE$(<string-exp>) Validates the string expression for a valid date according to the currently set DATEFORM (DATEFORM 3 is the same as DATEFORM 2). If the string is valid, the standard date format is created for that date. If the string is invalid, a null string is generated. A date may use any non-numeric character as a delimiter between month, day, and year except for the semicolon or comma.

```
Example:  PRINT DTE$("7/6/76"),DTE$("2/30/76"),DTE$("112154")
          07/06/76                      11/21/54
```

EXT$(<string-exp>,<num-exp1>,<num-exp2>) Returns with the subfield of the string expression whose position in string is indicated by the values of the two numeric expressions. The string returned is the subfield of the string whose position is the <num-exp2> subfield of <num-exp1> subfield.

```
Example:  B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

          PRINT EXT$(B$,2,0)
          BBBB

          PRINT EXT$(B$,3,3)
          C3C3C3

          PRINT EXT$(B$,3,0)
          C1C1C1]C2C2C2]C3C3C3

          PRINT EXT$(EXT$(B$,3,0),1,2)
          C2C2C2
```

As illustrated, when the second numeric expression is zero the entire field referenced by the first numeric expression is extracted. When the field designated by two numeric expressions does not exist in the string, a null string is returned.

Note: Field and subfield delimiters are not really the characters ^ and] because the parity bit is turned on to indicate that the character is a delimiter and not a normal ASCII character.

FORMAT$(<num-exp>,<string-exp>) This function has the same capabilies as the PRINT USING statement in regards to the formating of numeric values. The two expressions are evaluated and the numeric value is formated according to the masking characters in the string expression.

**	Leading asterisk fill.
$$	Leading floating dollar sign.
DB	Trailing literal of DB for negative values only.
CR	Trailing literal of CR for negative values only.
>	Number surrounded with angle brackets (<>) for negative values only.
#	Digit position with leading zero suppression.
9	Digit position with leading zero fill.
+	Trailing sign for positive and negative values.
-	Trailing minus sign for negative values only.
,,	Normalize number with commas every three digits.
^	Use exponential format with single unsigned digit exponent..
^^^	Use exponential format with signed single digit exponent.
^^^^	Use exponential format with signed double digit exponent.
^^^^^	Use exponential format with signed triple digit exponent.

```
Example:  PRINT FORMAT$(23,"99999");FORMAT$(23,"#####")
          00023   23

          PRINT FORMAT$(23,"**###");" ";FORMAT$(123456.78,"$$#,#
          ***23  $123,456.78

          PRINT FORMAT$(12345,"#.#####^^")
          1.2345E4

          PRINT FORMAT$(-12345.67,"#,######.##>")
           <12,345.67>
```

For more information and examples see the chapter "Formatted Input & Output" in this manual.

HEXOF$(<num-exp>) The numeric expression is evaluated, integerized and translated into the string of characters representing the value in hexadecimal. A four character string is always generated.

```
Example:  PRINT HEXOF$(94);" ";HEXOF$(23129)
          005E 5A59
```

INS$(<string-exp1>,<num-exp1>,<num-exp2>,<string-exp2>) This function is the inverse of the MID$ function, that is, it inserts a subfield into a string. The substring <string-exp2> will be inserted after the subfield designated by the values of the two numeric expressions. It is important to note that the field is inserted after the one designated.

```
Example:  B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

          PRINT INS$(B$,2,0,"NEW")
          AAAA^BBBB^NEW^C1C1C1]C2C2C2]C3C3C3^DDDD

          PRINT INS$(B$,0,0,"NEW")
          NEW^AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

          PRINT INS$(B$,3,1,"NEW")
          AAAA^BBBB^C1C1C1]NEW]C2C2C2]C3C3C3^DDDD

          PRINT INS$(B$,3,-1,"NEW")
          AAAA^BBBB^NEW]C1C1C1]C2C2C2]C3C3C3^DDDD

          PRINT INS$(B$,7,2,"NEW")
          AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD^^^^]]NEW
```

Note: Field and subfield delimiters are not really the characters ^ and] because the parity bit is turned on to indicate that the character is a delimiter and not a normal ASCII character.

LEFT$(<string-exp>,<num-exp>) Indicates a substring of the string expression from the first character through the nth character where n is the value of the numeric expression.

Example:
```
PRINT LEFT$(A$,7)
ABCDEFG
```

LPAD$(<string-exp>,<num-exp>) Adds leading spaces to a string. The two expressions are evaluated and the resulting string expression is expanded to the length indicated by the value of the numeric expression by adding sufficient leading spaces. If the string expression is already greater than or equal to the length indicated no spaces are added and the string is returned, unmodified.

Example:
```
PRINT "#";LPAD$("1234",6);"#"
#  1234#

PRINT "#";LPAD$("1234",3);"#"
#1234#
```

LTRIM$(<string-exp>) Removes leading spaces from a string. The string expression is evaluated and any leading spaces are removed.

Example:
```
PRINT LTRIM$("   ABC   DEF   ");"#"
ABC   DEF   #
```

MID$(<string-exp>,<num-exp1>,<num-exp2>) Indicates a substring of the string expression starting with character N1, for N2 characters where N1 and N2 are the values of the two numeric expressions. The length of the string returned will be at most N2-N1+1 characters.

Example:
```
PRINT MID$(A$,15,5)
OPQRS
```

OCTOF$(<num-exp>) The numeric expression is evaluated, integerized and translated into the string of characters representing the value of the number in octal. A six character string is always generated.

Example:
```
PRINT OCTOF$(123);" ";OCTOF$(94)
000175 000156
```

OVR$(<string-exp1>,<num-exp1>,<num-exp2>,<string-exp2>) Truncates or expands the second string expression to exactly N2 characters, where N2 is the value of the second numeric expression. Then the first string expression is overlaid by the second string expression, from position N1 for N2 characters.

Example:
```
PRINT OVR$(A$,2,3,"0123456")
A012EFGHIJKLMNOPQRSTUVWXYZ
```

REP$(<string-exp1>,<num-exp1>,<num-exp2>,<string-exp2>) This function is similar to the INS$ function except that it replaces a subfield instead of inserting the subfield. The substring <string-exp2> will replace the subfield designated by the values of the two numeric expressions. If there is no subfield to be replaced then the the substring will be inserted in its proper place. If the value of the second numeric expression is zero, the replacement is for the entire field designated by the first numeric expression.

If the first string expression does not have sufficient subfields, sufficient null fields will be added.

The string expression must not contain any characters whose value is greater than 127 or the results will be unpredicatable.

Using the character ^ as the field delimiter and] as the subfield delimiter:

Example:
```
B$ = AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD

PRINT REP$(B$,6,0,"HERE")
AAAA^BBBB^C1C1C1]C2C2C2]C3C3C3^DDDD^^HERE

PRINT REP$(B$,3,2,"NEW")
```

```
AAAA^BBBB^C1C1C1]NEW]C3C3C3^DDDD

PRINT REP$(B$,2,2,"NEW")
AAAA^BBBB]NEW^C1C1C1]C2C2C2]C3C3C3^DDDD
```

Note: Field and subfield delimiters are not really the characters ^ and] because the parity bit is turned on to indicate that the character is a delimiter and not a normal ASCII character.

RIGHT$(<string-exp>,<num-exp>) Returns the substring of the string expression from the nth character through the last character in the string expression where n is the value of the numeric expression.

```
Example: PRINT RIGHT$(A$,20)
         TUVWXYZ
```

RPAD$(<string-exp>,<num-exp>) Adds trailing spaces to a string. The two expressions are evaluated and the resulting string expression is expanded to the length indicated by the value of the numeric expression by adding sufficient trailing spaces. If the string expression is already greater than or equal to the length indicated no spaces are added and the string is returned, unmodified.

```
Example: PRINT "#";RPAD$("1234",6);"#"
         #1234  #

         PRINT "#";RPAD$("1234",3);"#"
         #1234#
```

RPT$(<num-exp>,<string-exp>) Generates a string of <num-exp> repetitions of the <string expression>.

```
Example: PRINT RPT$(3,"ABCD")
         ABCDABCDABCD
```

RTRIM$(<string-exp>) Removes trailing spaces from a string. The string expression is evaluated and any trailing spaces are removed.

```
Example: PRINT "#";RTRIM$("   ABC   DEF   ");"#"
         #   ABC   DEF#
```

SPACE$(<num-exp>) Returns a string of spaces of <num-exp> length.

```
Example: PRINT LEFT$(A$,3)&SPACE$(4)&MID$(A$,4,5)
         ABC    DEFGH
```

STR$(<num-exp>) **Indicates a string of numeric characters representing the value of the numeric expression. There are no leading or trailing blanks.**

```
Example: PRINT "ABC";STR$(1.23);"DEF"
         ABC1.23DEF

         PRINT "ABC";1.23;"DEF"
         ABC 1.23 DEF
```

TIME$(<num-exp>) Indicates a string of characters in normalized time format (i.e., hh:mm:ss) representing the numeric expression interpreted as the number of seconds since midnight of the current day. The value 0 (zero) is interpreted as the current time of day. The numeric expression must be in the range of 0 - 86399 or the function will return the time "00:00:00".

```
Example: PRINT TIME$(7199),TIME$(0)
         01:59:59          15:24:32
```

TRIM$(<string-exp>) Removes any leading or trailing spaces and reduces all embedded multiple spaces to a single space.

```
Example: PRINT "#";TRIM$("   ABC   DEF   HIJ   ");"#"
         #ABC DEF HIJ#
```

10.4 Input/Output Functions

INP Returns the ASCII, integer value of the first character of the last input, if the first character was a control character or a user defined key. When the first character was not a control character, the value of the function is 0. For example: if the last input was a CTRL/D the value of the INP is 4. If the last input was a CTRL/Z the value of INP is 26. If the last input was ABCDEFG the value of INP is 0.

Also see the appendix on "User Definable Keys".

LINE(<num-exp>) Returns the integer value of the ATTACHed line length of device opened on the I/O channel whose value is <num-exp>. I/O channel 0 may be used to indicate the console device.

```
Example:  PRINT LINE(0)  REM Console terminal
           79
```

PAGE(<num-exp>) Returns the integer value of the ATTACHed page length of device opened on the I/O channel whose value is <num-exp>. I/O channel 0 may be used to indicate the console device.

```
Example:  PRINT PAGE(0)  REM Console terminal
           23
```

POS(<num-exp>) Returns the integer count of the number of characters output on the I/O channel indicated by the numeric expression.

```
Example:  PRINT "123456";POS(0)
          123456 6
```

10.5 Logical Functions

The following functions allow the programmer to manipulate the bits of an integer value (binary word--16 bits). All of the arguments are numeric expressions whose value will be integerized.

LRL(<num-exp1>,<num-exp2>) If the value for either of the expressions is negative then it is replaced with the value 0. If the first exprression is greater than 65535 then it is replaced with the value 0. If the second expression is greater than 15 then it is replaced with the value 0. A logical rotate left is performed on the first integer for <num-exp2> bit positions.

LRR(<num-exp1>,<num-exp2>) If the value for either of the expressions is negative then it is replaced with the value 0. If the first expression is greater than 65535 then it is replaced with the value 0. If the second expression is greater than 15 then it is replaced with the value 9. A logical rotate right is performed on the value of <num-exp1> for <num-exp2> bit positions.

LSL(<num-exp1>,<num-exp2>) If the value for either of the expressions is negative then it is replaced with the value 0. If the first expression is greater than 65535 then it is replaced with the value 0. If the second expression is greater than 15 then it is replaced with the value 9. A logical shift left is performed on the value of <num-exp1> for <num-exp2> bit positions.

LSR(<num-exp1>,<num-exp2>) If the value for either of the expressions is negative then it is replaced with the value 0. If the first expression is greater than 65535 then it is replaced with the value 0. If the second expression is greater than 15 then it is replaced with the value 9. A logical shift right is performed on the value of <num-exp1> for <num-exp2> bit positions.

10.6 File Function

EOF(<num-exp>) The numeric expression is evaluated and the I/O channel corresponding to that value is checked for end-of-file condition. If the channel has not reached end-of-file the value of the function is 0 (false). If the channel has reached end-of-file the value of the function is -1 (true).

Channel zero (console) is never at end-of-file and will cause an error if tested with this function. Use the INP function to test for a CTRL/Z.

10.7 Error Functions

The following two functions do not have any arguments and should only be used in an error handling routine (see ON ERROR and RESUME statments).

ERL

Returns the integer line number of the statement causing the error to occur. A value of zero is returned if no error has occured.

Note: When this function is used on the left side of a relational expression and an unsigned integer is used on the right side of the same relational expression the RENUMBER command will assume that the unsigned integer is a line number and adjust it accordingly.

Note 2: When this function is used in a compiled program the value returned will be that of the hexadecimal address of the statement causing the error. This address is the same as that listed when the program was compiled. This should not affect the programmer except when the value of this function is displayed on the screen.

ERR

Returns the integer error number of the error that occured. A value of zero is returned if no error has occured. For a list of error numbers, their meanings and what might cause them see the appendix "Error Messages".

This function may be assigned a value with the LET statement in order that error handling routines may be tested.

10.8 USR Function

The USR function allows the BASIC programmer to interface a assembler language subroutine to the BASIC language program.

When the user requires a procedure to be accomplished that requires real-time processing or can only be done with the features of the CPU that are not available to the BASIC program, he must write an assembler language program. In many cases it is advantageous to only have a part of the procedure written in assembler code with the more routine processes accomplished with a BASIC language program. In order to transfer control and data between the user written subroutine and the BASIC program the USR function is used.

USR(<addr>,<num-exp>)

USR(<addr>,<string-exp>)

<addr> refers to the entry point address, relative to the load address of the assembler subroutine.

<num-exp> when evaluated and rounded to the nearest integer, is the sixteen bits of signed integer data to be transmitted to the assembler subroutine via the HL registers. When the subroutine is ready to return control back to BASIC the numeric value to be assigned to the function should be placed in the HL register pair.

<string-exp> when evaluated, is left in the "string accumulator". The address of this string accumulator is placed in the HL register pair before control is given to the user subroutine. The string accumulator is a 256 byte area that contains a one byte length followed by up to 255 characters. This area may be used by the subroutine as long as care is taken not to exceed the 256 byte limit. When the subroutine is ready to return control back to BASIC, it should load the HL register pair with the address of the string that is to be returned.

The USR function is a standard function call and may be used in any position of a BASIC statement that the other functions may be used.

Only one user written assembler language subroutine may be in memory while BASIC is executing, however the one subroutine may in fact be several subroutines concatenated together. Information may be found regarding assembly language programming in the OASIS MACRO Assembler Language Reference Manual.

The subroutine is loaded into memory by specifying it in an OPTION USR statement.

The OASIS MACRO Assembler Language Reference Manual has an example of a USR routine.

APPENDIX A

RESERVED WORDS

The following words are reserved and may not be used for variable names. The notation [...] means that a variable may not even start with the word indicated, if that variable is ever used with an implied LET statement.

ABS	FIX	MAX	RIGHT
AND	FLOAT	MID	RND
ASC	FNEND	MIN	ROUND
AT	FN[...]	MOD	RPAD
ATN	FOR	MOUNT	RPT
BIN	FORMAT	NBR	RTRIM
BINOF	GET	NEXT	RUN
CASE	GOSUB	OCT	SCH
CEND	GOTO	OCTOF	SEC
CHAIN	HEX	ON	SELECT
CHR	HEXOF	OPEN	SGN
CLEAR	IF	OPTION	SIN
CLOSE	IFEND	OR	SLEEP
COMMON	IMP	OTHERWISE	SPACE
COS	INP	OUTPUT	SQR
CRT	INPUT	OVR	STEP
CSI	INS	PAGE	STOP
DATA	INT	PI	STR
DATE	LEFT	POS	TAN
DAY	LEN	PRINT	THEN
DEF	LET	PROMPT	TIME
DEL	LINE	PUT	TRIM
DELETE	LINK	QUIT	UNLOCK
DIM	LINPUT	QUOTE	USR
DTE	LOG	RANDOMIZE	VAL
ELSE	LPAD	READ	WAIT
END	LRL	READNEXT	WEND
EOF	LRR	REM	WHILE
EQV	LSL	REM[...]	WRITE
ERL	LSR	REP	XOR
ERR	LTRIM	RESTORE	
EXP	MAT	RESUME	
EXT	MATCH	RETURN	

* All "variables" that start with the letters FN will <u>always</u> be treated as a reference to a user defined function. (See DEF statement.)

APPENDIX B

USER DEFINABLE KEYS

OASIS BASIC allows the programmer to code programs in such a manner that he can test whether certain keys were entered and then take whatever action he has programmed. These certain keys are the control keys, usually refered to by CTRL/x where x is one of the standard alphabetic keys modified by the control key.

When a program asks for keyboard input (MAT INPUT, INPUT, LINPUT, or LINPUT USING,) and the operator responds with a control key, program control will return to the BASIC program. The operator need not type a carriage return after the control key. No characters will be displayed on the console device when the operator types a control key.

The program can test which control key, if any, was entered by using the INP function. Only input from the console keyboard (I/O channel 0) may be tested with the INP function. The programmer may specify whatever action he wishes when the correct control key is entered.

This can be a very useful feature if the programmer is consistent in defining the meanings of the control keys. For instance he may define the CTRL/D to mean the current date. This is obviously easier for the operator to enter than typing the current date. It is also safer than programming a carriage return only to mean the current date or some other default value.

Some terminals have additional keys available to the operator. These are generally called function or program keys. If these keys generate an 8 bit code that is not one of the displayable ASCII characters then these keys may also be used as user definable keys by BASIC. The displayable ASCII characters have decimal values between 32 and 127, inclusive. To determine the exact values generated by these keys refer to the operators or users manual for the specific terminal.

Example:

The following is a simple program that shows the user definable key feature of OASIS BASIC.

```
10     OPTION PROMPT CHR$(0)
20 LOOP: PRINT "Please type a control key: ";
30     LINPUT USING "!",KEY$\PRINT
40     IF INP=0 THEN IF KEY$<>"" THEN GOTO ERROR
50     PRINT "The key you typed has a value of";INP;
60     PRINT "and was the key ";CHR$(INP+64)
70     GOTO LOOP
80 ERROR: PRINT "You don't follow directions very well."
90     GOTO LOOP
```

B.1 Control Key Values

Key	Value		Key	Value		Key	Value		Key	Value	
@	0		H	8	*	P	16		X	24	
A	1		I	9	*	Q	17		Y	25	
B	2		J	10		R	18		Z	26	
C	3		K	11		S	19		[	27	**
D	4	*	L	12		T	20		\	28	
E	5		M	13		U	21		]	29	
F	6	*	N	14		V	22		^	30	
G	7		O	15		W	23		_	31	***

* These key values are used for editing by LINPUT USING, and/or INPUT statments.

** This is the escape code. Because the system control keys are escape sequences entry of this key once is an indication to OASIS that the next character may be a system request. To get a value 27 passed to the INP function the operator must type this key twice. When this is done one escape character is passed to BASIC which, if it is the first character of an input field, will set the INP function to 27.

*** This value may also bew generated by some terminals by a CTRL/DEL or CTRL/RUB.

Some systems have other keys that may be tested by this function. If this situation is possible then you should use the above program to detect and determine

the value of the specific keys.

It is possible that a particular system may have other or different keys that are trapped by the operating system and never passed to the BASIC program. It is also possible that some keys may generate different values than those listed here. Both of these situations are dependent upon the SET values for: RUBOUT, LEFT, RIGHT, UP, DOWN, CANCEL, ESCAPE and the console class code. For more information see the chapter "SET COMMAND", in the OASIS System Reference Manual.

APPENDIX C

COMMAND SUMMARY

AUTO

AUTO [<start line>[,<increment value>]]

BOTTOM

BOTTOM

BREAK

BREAK [AT <line reference> [AFTER <count>]]
BREAK [ON <variable> [CHANGE] [AFTER <count>]]
BREAK [ON <variable> <relation> <value>]

CHANGE

CHANGE /from string/to string/ [<range>]

CONTINUE

CONTINUE

DELETE

DELETE [<range>]

DOWN

<line-feed>
<down arrow key>

HELP

HELP

INDENT

INDENT [<indent value>]

LENGTH

LENGTH

LIST

LIST [<range>]
<carriage return>

LOAD

LOAD <program name> [<program type>]

LOCATE

LOCATE /<string>/ [<range>]

LPLIST

LP[n]LIST [<range>]

LPXREF

LP[n]XREF

MODIFY

MODIFY [<range>]

NAME

NAME [<program name>[.<program type>[:<program disk>]]]

NEW

NEW

QUIT

QUIT [<number>]
QUIT [<unquoted string>]

RENUMBER

RENUMBER [<first> [<incr> [<start> [<end>]]]]

RUN

RUN [<program name>] [<starting line>]

SAVE

SAVE [<program name> [<program type> [<program disk>]]]

STEP

STEP [<count>]

TOP	TOP
TRACE	TRACE TRACE VARS
UNBREAK	UNBREAK [AT <line reference>] UNBREAK [ON <variable>]
UNTRACE	UNTRACE
UP	<up-arrow key> <control/Z>
VARS	VARS [<variable list>]
XREF	XREF

APPENDIX D

STATEMENT SUMMARY

CASE
```
[<line-no>] [<label>] CASE <expression>
```

CEND
```
[<line-no>] [<label>] CEND
```

CHAIN
```
[<line-no>] [<label>] CHAIN <string expression>
```

CLEAR
```
[<line-no>] [<label>] CLEAR [<variable list>]
```

CLOSE
```
[<line-no>] [<label>] CLOSE #<channel>
```

COMMON
```
[<line-no>] [<label>] COMMON <variable list>
```

CSI
```
[<line-no>] [<label>] CSI <string expression>
```

DATA
```
[<line-no>] [<label>] DATA <literal>[,<literal>]...
```

DEF
```
[<line-no>] [<label>] DEF FN<simple variable>[(<arg list>)] [= <expression>]
```

DELETE
```
[<line-no>] [<label>] DELETE #<channel>,<key>
```

DIM
```
[<line-no>] [<label>] DIM <simple var>(<numeric expr>[,<numeric expr>])>...
```

ELSE
```
[<line-no>]           ELSE <statement>
[<line-no>]           ELSE <line number>
```

END
```
[<line-no>] [<label>] END
```

FNEND
```
[<line-no>] [<label>] FNEND
```

FOR
```
[<line-no>] [<label>] FOR <num var>=<num exp> TO <num exp>[ STEP <num exp>]
[<line-no>] [<label>] FOR <var> = <literal list>
```

GET
```
[<line-no>] [<label>] GET <device> <numeric expr>,<variable list>
```

GOSUB
```
[<line-no>] [<label>] GOSUB <line reference>
[<line-no>] [<label>] GO SUB <line reference>
```

GOTO
```
[<line-no>] [<label>] GOTO <line reference>
[<line-no>] [<label>] GO TO <line reference>
```

IF
```
[<line-no>] [<label>] IF <relation> THEN <statement> [ELSE <statement>]
[<line-no>] [<label>] IF <relation> THEN <line-ref> [ELSE <line-ref>]
[<line-no>] [<label>] IF <rel>
```

IFEND
```
[<line-no>] [<label>] IFEND
```

INPUT
```
[<line-no>] [<label>] INPUT [<prompt expression>,]<variable list>
[<line-no>] [<label>] INPUT #<channel>:<variable list>
[<line-no>] [<label>] INPUT #<channel>,<key>:<variable list>
```

LET

```
[<line-no>] [<label>] [LET] <string variable> = <string expression>
[<line-no>] [<label>] [LET] <numeric var> = <numeric expr>
[<line-no>] [<label>] [LET] <string variable><substring> = <string expr>
[<line-no>] [<label>] [LET] <user defined function> = <expression>
[<line-no>] [<label>] [LET] ERR = <numeric expression>
```

LINK

```
[<line-no>] [<label>] LINK <string expression>
```

LINPUT

```
[<line-no>] [<label>] LINPUT [<prompt expression>,]<string variable>
[<line-no>] [<label>] LINPUT #<channel>:<string variable>
[<line-no>] [<label>] LINPUT #<channel>,<key>:<string variable>
[<line-no>] [<label>] LINPUT [<prompt expr>,] USING <mask>,<string var>
```

MAT

```
[<line-no>] [<label>] MAT <array name> = <array name>
[<line-no>] [<label>] MAT <array name> = (<expression>)
```

MAT INPUT

```
[<line-no>] [<label>] MAT INPUT <array name>
[<line-no>] [<label>] MAT INPUT #<channel>: <array name>
[<line-no>] [<label>] MAT INPUT #<channel>,<key>: <array name>
```

MAT PRINT

```
[<line-no>] [<label>] MAT PRINT <array name list> <punct>
[<line-no>] [<label>] MAT PRINT #<channel>: <array name list> <punct>
[<line-no>] [<label>] MAT PRINT #<channel>,<key>: <array name list> <punct>
```

MAT READ

```
[<line-no>] [<label>] MAT READ <array name>
[<line-no>] [<label>] MAT READ #<channel>: <array name>
[<line-no>] [<label>] MAT READ #<channel>,<key>: <array name>
```

MAT WRITE

```
[<line-no>] [<label>] MAT WRITE #<channel>: <array name>
[<line-no>] [<label>] MAT WRITE #<channel>,<key>: <array name>
```

MOUNT

```
[<line-no>] [<label>] MOUNT <string expression>
```

NEXT

```
<line-no> NEXT [<variable>]
```

ON ERROR

```
[<line-no>] [<label>] ON ERROR GOTO <line reference>
[<line-no>] [<label>] ON ERROR GOTO 0
```

ON

```
[<line-no>] [<label>] ON <numeric expression> GOTO <line reference list>
[<line-no>] [<label>] ON <numeric expression> GOSUB <line reference list>
```

OPEN

```
[<line-no>] [<label>] OPEN #<channel>: <string expr>,<mode> <method>[<options>]
```

OPTION

```
[<line-no>] [<label>] OPTION <option list>
```

OTHERWISE

```
[<line-no>] [<label>] OTHERWISE
```

PRINT

```
[<line-no>] [<label>] PRINT [<expression list><punct>]
[<line-no>] [<label>] PRINT #<channel>[:<expression list><punct>]
[<line-no>] [<label>] PRINT #<channel>,<key>[:<expression list><punct>]
```

PRINT USING

```
[<line-no>] [<label>] PRINT USING <mask>,<expression list><punct>
[<line-no>] [<label>] PRINT #<channel>: USING <mask>,<expr list><punct>
[<line-no>] [<label>] PRINT #<channel>,<key>: USING <mask>,<expr list><punct>
```

PUT

```
[<line-no>] [<label>] PUT <device> <numeric expression>,<expression list>
```

QUIT
```
    [<line-no>] [<label>] QUIT [<expression>]
```
RANDOMIZE
```
    [<line-no>] [<label>] RANDOMIZE
```
READ
```
    [<line-no>] [<label>] READ <variable list>
    [<line-no>] [<label>] READ #<channel>: <variable list>
    [<line-no>] [<label>] READ #<channel>,<key>: <variable list>
```
READNEXT
```
    [<line-no>] [<label>] READNEXT #<channel>,<string key>: <variable list>
```
REM
```
    [<line-no>] [<label>] REM <any characters>
```
RESTORE
```
    [<line-no>] [<label>] RESTORE [<line number>]
```
RESUME
```
    [<line-no>] [<label>] RESUME <line reference>
    [<line-no>] [<label>] RESUME 0
```
RETURN
```
    [<line-no>] [<label>] RETURN [<line ref>]
```
RUN
```
    [<line-no>] [<label>] RUN [<string expression>]
```
SELECT
```
    [<line-no>] [<label>] SELECT [<expression>]
```
SLEEP
```
    [<line-no>] [<label>] SLEEP <numeric expression>
```
STOP
```
    [<line-no>] [<label>] STOP [<expression>]
```
THEN
```
    [<line-no>]           THEN <statement>
    [<line-no>]           THEN <line number>
```
UNLOCK
```
    [<line-no>] [<label>] UNLOCK #<channel>
```
WAIT
```
    [<line-no>] [<label>] WAIT
    [<line-no>] [<label>] WAIT DEVICE <numeric expression>
    [<line-no>] [<label>] WAIT PORT <numeric expr>,<numeric expr>[,<numeric expr>]
    [<line-no>] [<label>] WAIT MEMORY <numeric exp>,<numeric exp>[,<numeric exp>]
```
WEND
```
    [<line-no>] [<label>] WEND
```
WHILE
```
    [<line-no>] [<label>] WHILE <numeric expression>
```
WRITE
```
    [<line-no>] [<label>] WRITE #<channel>: <expression list>
    [<line-no>] [<label>] WRITE #<channel>,<key>: <expression list>
```

APPENDIX E

FUNCTION SUMMARY

In the following summary the arguments N, N1, and N2 all represent numeric expressions; the arguments A$ and B$ all represent string expressions.

ABS(N)	Returns the absolute value of N.
ASC(A$)	Returns the ASCII value of the first character in A$.
AT$(N1,N2)	Returns the string of characters that, if printed, would position the cursor at N1,N2. (N1 is horizontal, N2 is vertical.)
ATN(N)	Returns the arctangent of N (N in radians).
BIN(A$)	Returns a decimal value for the binary A$.
BINOF$(N)	Returns a string representing the binary value of N.
CHR$(N)	Returns the character having the ASCII value of N.
COS(N)	Returns the cosine of N (N in radians).
CRT$(A$)	Performs non x/y console output control.
DATE$(N)	Internal date to external date.
DAY(A$)	External date to internal date.
DEL$(A$,N1,N2)	Returns A$ with field designated by N1 and N2 removed.
DTE$(A$)	Test A$ for valid date. When valid converts to normalized format, else returns null string.
EOF(N)	Returns End-Of-File flag for I/O channel N.
ERL	Returns line number of statement causing error.
ERR	Returns number of error.
EXP(N)	Returns the value of e^N.
EXT$(A$,N1,N2)	Returns the substring of A$ for field N1 of A$ and subfield N2 of field N1.
FIX(N)	Returns the integerized value of N.
FLOAT(N)	Converts integer N to floating point.
FORMAT$(N,A$)	Formats N according to mask A$.
HEX(A$)	Returns a decimal value for the hexadecimal A$.
HEXOF$(N)	Returns a string representing the hexadecimal value of N.
INP	Returns the numeric value of the control character input to console.
INS$(A$,N1,N2,B$)	Returns the string of A$ with string B$ inserted after the substring of A$ for field N1 of A$ and subfield N2 of field N1.
INT(N)	Returns the greatest integer which is less than or equal to N.
LEFT$(A$,N)	Returns A$ from the first character to the Nth character.
LEN(A$)	Returns the length of string A$.
LINE(N)	Returns line length of device opened on channel N.
LOG(N)	Returns the natural logarithm of N.
LPAD$(A$,N)	Adds leading spaces to A$ to make string of length N.
LRL(N1,N2)	Logical rotate left N1 for N2 bit positions.
LRR(N1,N2)	Logical rotate right N1 for N2 bit positions.

LSL(N1,N2)	Logical shift left N1 for N2 bit positions.
LSR(N1,N2)	Logical shift right N1 for N2 bit positions.
LTRIM$(A$)	Remove leading spaces from string A$.
MATCH(A$,B$)	Tests string A$ against mask B$; returns true/false (-1/0).
MAX(N1,N2)	Returns the greater value of N1 and N2.
MID$(A$,N1,N2)	Returns A$ from the N1th character for N2 characters.
MIN(N1,N2)	Returns the lessor value of N1 and N2.
MOD(N1,N2)	Returns remainder of N1 divided by N2.
NBR(A$)	Test A$ for numerics. Returns 0 if any non-numeric characters in A$, else returns -1.
OCT(A$)	Returns a decimal value for the octal A$.
OCTOF$(N)	Returns a string representing the octal value of N.
OVR$(A$,N1,N2,B$)	Returns A$ with B$ overlaid, starting at N1th character for N2 characters.
PAGE(N)	Returns page length of device opened on channel N.
PI	Returns the constant value 3.141592653590.
POS(N)	Returns the current character position of output channel N.
REP$(A$,N1,N2,B$)	Returns the string of A$ with string B$ replacing the substring of A$ for field N1 of A$ and subfield N2 of field N1.
RIGHT$(A$,N)	Returns A$ from the Nth character to the end.
RND	Returns a random number between 0 and 1, exclusive.
ROUND(N1,N2)	Rounds N1 to number of positions indicated by N2.
RPAD$(A$,N)	Adds trailing spaces to A$ to make string of length N.
RPT$(N1,A$)	Returns the string of N1 repetitions of A$.
RTRIM$(A$)	Removes trailing spaces from string A$.
SCH(N1,A$,B$)	Returns the character position of the string B$ within A$ with the search starting at character position N1.
SEC(A$)	External time to internal time.
SGN(N)	Returns the algebraic sign of N (+ or -).
SIN(N)	Returns the sine of N (N in radians).
SPACE$(N)	Returns the string of N blanks.
SQR(N)	Returns the square root of N.
STR$(N)	Returns the string of characters representing the number N.
TAN(N)	Returns the tangent of N (N in radians).
TIME$(N)	Internal time to external time.
TRIM$(A$)	Remove leading and trailing spaces from string A$.
USR(N1,N2)	Calls assembly subroutine at relative location N1, passing N2 to the routine.
USR$(N,A$)	Cals assembly subroutine at relative location N, passing A$ to the routine.
VAL(A$)	Returns the numeric value of A$.

APPENDIX F

RUN2 STATEMENT AND FUNCTION EXCEPTIONS

RUN2, the smaller version of the OASIS BASIC, performs exactly like the standard RUN command except that certain statements and functions of OASIS BASIC have been omitted. Omitting these features reduces the overhead requirements of BASIC by approximately 3 thousand bytes of memory. As can be seen the statements and functions that have been omitted from RUN2 are not normally used by most application programs and will cause no problems for most users.

When a program is executed using RUN2 and an attempt is made to execute one of the statements or functions that have been omitted from RUN2 a non-trappable error 44 occurs.

F.1 Statements Omitted

The statements that have been omitted from RUN2 are: GET, PUT, RANDOMIZE, and WAIT.

F.2 Functions Omitted

The functions that have been omitted from RUN2 are: ATN, BIN, BINOF$, COS, DEL$, EXP, EXT$, HEX, HEXOF$, INS$, LOG, LRL, LRR, LSL, LSR, OCT, OCTOF$, REP$, RND, SIN, and TAN.

APPENDIX G

ERROR MESSAGES

G.1 Command Errors

AUTO cannot replace or merge lines

Indicates that the AUTO command attempted to use a line number already in use or that there was a line whose line number was between the last auto line number and the next auto line number to be used.

Disk Full

Indicates that the disk used by the SAVE command is full. Remember that saving an existing file causes the previous version of the file to be renamed BACKUP.

Insufficient Memory

An attempt was made to add another line to the program in memory that could not fit into the available memory.

Invalid command syntax

Indicates that the command was recognized but a syntax error was detected.

Invalid Program Name

An attempt was made to NAME, SAVE, LOAD, or COMPILE a program using an invalid name. The program name must be at least two characters in length and start with a letter.

Invalid Statement Number

An attempt was made to enter or display a line with an invalid line number. Line numbers must be between 1 and 9999.

Renumber Range Error

Indicates that the line numbers that would be generated by the RENUMBER command would cause lines to change their relative location in the program.

String missing or invalid

Occurs on a CHANGE or LOCATE command when no previous CHANGE or LOCATE command has been executed and no valid string arguments were specified.

Unrecognized command

Indicates that the command name was abbreviated too much or misspelled to an extent that the command desired could not be discerned.

G.2 Edit Errors

Comma Required

Colon Required

End of Line Required

Equal Sign Required

Expression Required

File Mark Required

Keyword Missing or mis-spelled

Missing Parenthesis

Numeric Expression Required

Numeric Variable Required

Statement Number Required

String Expression Required

String Variable Required

Terminating Quote Required

Too Many Subscripts

Unbalanced Parenthesis

Unrecognized Statement

G.3 Compile Errors

CASE without SELECT.

CASE statements may only be used within a SELECT-CASE-CEND structure.

CASEless SELECT.

Each SELECT-CASE-CEND structure must have at least one CASE statement.

CASE following OTHERWISE.

CASE statements may not follow an OTHERWISE statement as it will not be executed.

CEND without SELECT.

CEND statements may only be used to denote the end of a SELECT-CASE-CEND structure.

ELSE without IF.

ELSE statements may only be used in a multi-line IF-IFEND structure.

FNEND without DEF.

FNEND statements may only be used to denote the end of a multi-line function definition.

FNEND missing.

FNEND statements must be used to denote the end of a multi-line function definition.

FOR without NEXT.

Every FOR statement must have one matching NEXT statement following it in the program.

FOR nested too deep.

FOR-NEXT structures may only be nested to 32 levels.

IF nested too deep.

IF-IFEND structures may only be nested to 32 levels.

IFEND without IF.

IFEND statements may only be used to denote the end of an IF-IFEND structure.

Illegal DEF nesting.

Multi-line function definitions may not be nested.

IF without IFEND.

Every multi-line IF-IFEND structure must have one matching IFEND statement following the IF in the program.

Label is multi defined.

Line labels may only be defined once in a program.

More than one OTHERWISE.

A SELECT-CASE-CEND structure may only have one OTHERWISE statement.

More than one ELSE.

An IF-IFEND structure may only have one ELSE statement.

NEXT without FOR.

The NEXT statement may only be used to denote the end of a FOR-NEXT structure and the NEXT statement must physically follow the FOR statement in the program.

OTHERWISE without SELECT.

The OTHERWISE statement may only be used within a SELECT-CASE-CEND structure.

Reference to undefined line

Line numbers referenced in a program must exist in the program.

Reference to undefined label

Line labels referenced in a program must be defined in the program.

RUN, LINK, or CHAIN has line number.

The line number operand of the RUN, CHAIN, and LINK statements is no longer available.

SELECT nested too deep.

SELECT-CASE-CEND structures may only be nested to a level of 32 deep.

SELECT without CEND.

SELECT-CASE-CEND structures must be terminated with a CEND statement.

Too long

A line may not exceed the 253 character limit during compilation.

WEND without WHILE.

WEND statements may only be used to denote the end of a WHILE-WEND structure.

WHILE's nested too deep.

WHILE-WEND structures may only be nested 32 levels deep.

WHILE without WEND.

WHILE-WEND structures must be terminated with a WEND statement.

G.4 Execution Errors

The following errors may occur during the execution of a program. They are all trappable by user written error routines unless stated otherwise. In general, the non-trappable errors indicate a programming logic error that could not be corrected at run time anyway.

1 ESC-C

Operator typed an ESC,C during execution of the program.

2 Divide by Zero

Occurs during expression analysis if an attempt is made to divide by zero.

3 Overflow

An integer expression resulted in a value outside the range -32767 to +32767 or a floating point exprression resulted in a value outside the range of -10^126 to +10^126.

4 Underflow

A floating point expression resulted in a value outside the range of -10^-126 to +10^-126.

5 Illegal Number

Occurs on input type statements or string to numeric conversion type functions when the string of characters contains characters that are not allowed in numeric fields.

6 SQR of Negative

7 LOG of Zero

8 LOG of Negative

9 Insufficient Memory (non-trappable)

Occurs during execution when a statement attempts to define additional working storage that exceeds the amount of memory available.

10 Line not Found (non-trappable)

Occurs on the statements: CHAIN, ELSE, GOSUB, GOTO, LINK, ON ERROR, ON, RESTORE, RESUME, RETURN, RUN, or THEN when the line number specified is not used in the program.

11 Label not Found (non-trappable)

Occurs on the statements: ELSE, GOSUB, GOTO, ON ERROR, ON, RESUME, and RETURN, when the line label specified is not defined in the program.

12 Return Stack Empty (non-trappable)

Occurs on the RETURN statement when there is no GOSUB in effect.

13 WEND without WHILE (non-trappable)

Occurs on the WEND statement when there is no WHILE in effect. (A WHILE statement without a WEND is okay because the end of the program is encountered.)

14 NEXT without FOR (non-trappable)

Occurs on the NEXT statement when there is no FOR in effect. (A FOR statement without a NEXT is okay because the end of the program is encountered.)

15 Insufficient Data

Occurs on the INPUT statement when multiple fields are to be input and fewer fields are actually entered.

16 Invalid File Number (non-trappable)

Can occur on any of the file I/O statements when the channel number expression is less than 1 or greater than 16. Can also occur on any of the file functions.

17 RESUME without Error (non-trappable)

Occurs on the RESUME statement when there is no error in effect.

18 Invalid Address (non-trappable)

Can occur on any of the statements or functions that access memory when the address is out of range.

19 Invalid Separator

Occurs on input statements.

20 ON Range Error

Occurs in the ON GOSUB or ON GOTO statement when the numeric expression is less than one or greater than the number of line references specified.

21 CEND without SELECT (non-trappable)

Occurs on the CEND statement when there is no SELECT in effect.

22 Type Mismatch (non-trappable)

Occurs during file reads when the variable type requested does not match the variable type actually read, or in a SELECT structure when the expression type of the expression selected does not match the type of the case tested.

23 Invalid Zero Dimension (non-trappable)

Occurs if an OPTION BASE 1 has been executed and a reference is made to the zero subscript of an array.

24 Inconsistent Usage (non-trappable)

Occurs when a DIM or COMMON attempts to dimension a array with the same name as a varible already defined or after an array is dimension and a reference is made to the same name in a variable.

25 Subscript Range (non-trappable)

Occurs on any reference to a subscripted array less than or greater than the number of elements dimensioned in the array.

26 Invalid Using (non-trappable)

Indicates that a PRINT USING statement mask specified string when the expression field was numeric or the mask specified numeric when the field was string.

27 File is Closed

Occurs on an attempt to CLOSE an I/O channel that is not currently open.

28 File is Open

Occurs on an attempt to OPEN an I/O channel that is currently in use.

29 Invalid File Name

Occurs on the OPEN, CHAIN, RUN, or LINK statement when the file or program name is invalid. File and program names must start with a letter. Program file types must be BASIC or BASICOBJ. Can also occur on the OPEN statement when the device name is mis-spelled.

30 File not Found

31 Disk Full

Indicates an attempt was made to add more data to the disk when there wasn't sufficient space available on the disk. Can only occur on sequential file format output.

32 Directory Full

33 Protected File

Indicates an attempt was made to OPEN a file that was read protected, or an attempt was made to re-create a file that was delete protected, or an attempt was made to output to a file that was write protected.

34 Invalid Key

Indicates that an input or output type statement used a key on a sequential format file or did not use the proper type of key for a direct or indexed format file.

35 Wrong Access

Occurs on input type statements when the file was opened for output or on an output type statement when the file was opened for input.

36 Out of DATA

Occurs on the READ statement when there are no more DATA elements.

37 OPTION BASE must precede DIM (non-trappable)

Occurs on the OPTION BASE statement when there are variables defined. Should perform a CLEAR first.

38 No USR Program (non-trappable)

Indicates a reference to a USR function when there is no USR program loaded.

39 Invalid Drive Code (non-trappable)

Occurs on the MOUNT statement when the drive is invalid or not attached.

40 Program not Found (non-trappable)

Occurs on a CHAIN, LINK, or RUN statement if the specified program cannot be found.

41 Invalid File Format (non-trappable)

Occurs on any attempt to input from a file created prior to version 5.3I OASIS and not converted with the FILECONV program; using READ on a record that was output with a PRINT; or using INPUT on a record that was output with a WRITE.

42 FNEND without DEF (non-trappable)

Occurs when an FNEND statement is encountered outside of a user defined function definition.

43 DEF not found (non-trappable)

Occurs on any reference to a user defined function that is not defined in the current program.

44 Unimplemented feature (non-trappable)

Occurs during execution with RUN2 when a reference is made to a statement or function that is not available with the smaller version of OASIS BASIC.

45 File Full (non-trappable)

Occurs during an attempt to write more records to an indexed file than it can hold.

46 Device not Attached (non-trappable)

Occurs during an attempt to OPEN a channel to a device that is not attached.

APPENDIX H

PROGRAM EXAMPLES

H.1 Example One

```
-10   INPUT "Radius of circle",R
-20   PRINT "Diameter =";2.*R
-30   PRINT "Area =";PI*R^2.
-40   PRINT "Circumference =";2.*PI*R
-50   END
-RUN

Radius of circle? 2.5
Diameter = 5
Area = 19.63495408493
Circumference = 15.70796326795
```

H.2 Example Two - String Conversion

The following example illustrates a method of translating the individual characters of a string into the decimal equivalents.

```
-AUTO
  10      DIM X(3)
  20      LET STRING$ = "CAT"
  30      X(0) = LEN(STRING$)
  40      FOR I% = 1 TO X(0)
  50           X(I%) = ASC(MID$(STRING$,I%,1))
  60           NEXT I%
  70      PRINT "STRING$ = ";STRING$,"Length of STRING$ =";X(0)
  80      PRINT X(0),X(1),X(2),X(3)
  90      END
 100
-RUN

STRING$ = CAT       Length of STRING$ = 3
 3                   67                  65                  84
```

H.3 Example Three - Sine Wave

This example will produce a sine wave on the console terminal.

```
10 MAGNITUDE% = LINE(0)*3./8.
20 MIDDLE% = LINE(0)/2.
30 FREQUENCY = .175
40 FOR J = 0 TO 100 STEP FREQUENCY
50 SINE = INT(MAGNITUDE%*SIN(J))
60 PRINT SINE;TAB(MIDDLE%+SINE);"*"
70 NEXT
```

H.4 Example Four - Bill of Materials

```
-AUTO
  10 REM Accept Bill of Materials
  20       PRINT "How many items";
  30       INPUT ITEM.COUNT%
  40       FOR I% = 1 TO ITEM.COUNT%
  50            INPUT Q(I%),P(I%)
  60            NEXT
  70       PRINT
  80 REM Display Bill of Materials, extension, and total
  90       PRINT "Item          Quantity       Price        Amount"
 100       PRINT
 110       MASK1$=" ##"&SPACE$(11)&"#####"&SPACE$(9)&"$###.##"&SPACE$(7)&"$#,###.##"
 120       FOR I% = 1 TO ITEM.COUNT%
 130            PRINT USING MASK1$,I%,Q(I%),P(I%),Q(I%)*P(I%)
 140            LET TOTAL = TOTAL+Q(I%)*P(I%)
 150            NEXT
 160       PRINT
 170       LET MASK2$ = "  TOTAL"&SPACE$(35)&"$#,###.##"
 180       PRINT USING MASK2$,TOTAL
 190 END
 200
-RUN

How many items? 5
? 2,750
? 25,23.50
? 10,85.35
? 145,.08
? 75,2.35

Item          Quantity      Price         Amount

  1                2         $750.00       $1,500.00
  2               25         $ 23.50       $  587.50
  3               10         $ 85.35       $  853.50
  4              145         $  0.08       $   11.60
  5               75         $  2.35       $  176.25

  TOTAL                                    $3,128.85
```

H.5 Example Five

The following sample illustrates the use of three functions (INP, INS, REP) and uses a disk file.

```
-AUTO
  10 OPEN #1: "NAME.DATA:A",OUTPUT SEQUENTIAL\REM Create file
  20 PRINT CRT$("C");              \REM Clear screen
  30 PRINT AT$(1,6); "Name";       \REM Display all the input fields
  40 PRINT AT$(1,7);"Address";
  50 PRINT AT$(1,8);"City";
  60 PRINT AT$(33,8);"State";
  70 PRINT AT$(53,8);"Zip";
  80 PRINT AT$(1,9);"SSN";
  90 OPTION PROMPT ":"
 100 R$ = ""                       \REM Initialize record
 110 PRINT AT$(5,6);               \REM Position for first input field
 120 LINPUT USING "                    ", A$
 130 IF INP = 26 THEN 999          \REM Entry of CTRL/Z means end
 140 R$ = INS$(R$,0,0,A$)          \REM A$ is first field
 150 PRINT AT$(8,7);               \REM Position for second input field
 160 LINPUT USING "                    ", A$
 170 R$ = INS$(R$,1,0,A$)          \REM A$ is second field
 180 PRINT AT$(5,8);               \REM Position for third field
 190 LINPUT USING "                ", A$
 200 R$ = INS$(R$,2,0,A$)          \REM A$ is third field
 210 PRINT AT$(38,8);              \REM Position for fourth field
 220 LINPUT USING "  ", A$
 230 R$ = INS$(R$,3,1,A$)          \REM A$ is second subfield of third
 240 PRINT AT$(56,8);              \REM Position for fifth field
 250 LINPUT USING "     ", A$
 260 R$ = REP$(R$,3,3,A$)          \REM A$ is subfield 3 of third field
 270 PRINT AT$(4,9);               \REM Position for sixth field
 280 LINPUT USING "            ", A$
 290 R$ = REP$(R$,4,0,A$)          \REM A$ is fourth field
 300 PRINT #1:R$                   \REM Create new record in file
 310 GOTO 20                       \REM Start over
 320 END
 330
```

Lines 20 through 80, when executed, will display the field names:

```
Name
Address
City                                    State               Zip
SSN
```

Lines 110 through 280, when executed, position the cursor after each field name and allow input. Each field input is saved in the string R$ with appropriate field delimiters. For instance:

```
Name: JOSEPH E. BROWN
Address: 1234 S.E. MAIN STREET
City: SAN FRANCISCO                     State: CA           Zip: 99999
SSN: 123-45-6789
```

The above entry will produce a record that looks like this:

JOSEPH E. BROWN^1234 S.E. MAIN STREET^SAN FRANCISCO]CA]99999^123-45-6789

The characters ^ and] are the field delimiters. They differ from the normal ASCII character by having the parity bit turned on.

H.6 Example Six - Sequential File I/O

The following example illustrates file input and formatted output using the AT function.

```
-AUTO
  10 OPEN #1: "NAME.DATA",INPUT SEQUENTIAL
  20 LOOP: PRINT CRT$("C");
  30           I% = 0
  40           INPUT: LINPUT #1: A$
  50                IF EOF(1) THEN GOTO EXIT
  60                PRINT AT$(6,I%+3);EXT$(A$,1,0);
  70                PRINT AT$(6,I%+4);EXT$(A$,2,0);
  80                PRINT AT$(6,I%+5);EXT$(A$,3,1);\L = LEN(EXT$(A$,3,1))
  90                PRINT AT$(8+L,I%+5);EXT$(A$,3,2);\L = L+LEN(EXT$(A$,3,2))
 100                PRINT AT$(10+L,I%+5);EXT$(A$,3,3);
 110                PRINT AT$(6,I%+6);EXT$(A$,4,0);
 120                I% = I%+5 \ IF I%+5 < 23 THEN GOTO INPUT
 130           WAIT
 140           GOTO LOOP
 150 EXIT: END
 160
-
```

Assuming that the file "NAME" contains the record from Example 5 the display will be as follows:

```
JOSEPH E. BROWN
1234 S.E. MAIN STREET
SAN FRANCISCO  CA  99999
123-45-6789
```

There will be four names per page with two blank lines separating each name from the next.

H.7 Example Seven - Indexed File I/O - Sequential Access

The following example illustrates a simple sequential list to the primary printer of a name and address file, printing in label format.

The format of the file being read is: Key = name, last name first, separated by a comma, space from first name; record = address, city, state, zip, etc.

```
-AUTO 1000
1000 OPEN #1: "NAMES.ADDRESS", INPUT INDEXED
1010 OPEN #2: "PRINTER1", OUTPUT SEQUENTIAL, FORMAT
1020 READ: READNEXT #1,KEY$: ADDR$,CITY$,STATE$,ZIP$ \REM Get next record
1030       IF EOF(1) THEN 1170                \REM At end?
1040       C=SCH(KEY$,1," ")                  \REM Find end of last name
1050       IF NOT C THEN 1070                 \REM Not found - assume okay
1060       KEY$=RIGHT$(KEY$,C+2)&LEFT$(KEY$,C-1) \REM Restructure name
1070       PRINT #2: " ";KEY$                 \REM Print the name
1080       IF LEN(ADDR$)=0 THEN 1100          \REM If no address skip
1090       PRINT #2: " ";ADDR$
1100       PRINT #2: " ";CITY$;" ";STATE$;"  "; \REM Print city and state
1110       IF LEN(ZIP$)=5 THEN 1130           \REM If zip full then skip
1120       ZIP$=FORMAT$(VAL(ZIP$),"99999") \REM Format Zip
1130       PRINT #2: ZIP$                     \REM Print the zip code
1140       IF LEN(ADDR$)=0 THEN PRINT #2: " " \REM Account for lost line
1150       PRINT #2: "-"                      \REM Triple space for next 'label'
1160       GOTO READ                          \REM Get another record
1170 REM End of file - clean up
1180       CLOSE #1 \ CLOSE #2 \QUIT
1190 END
```

In addition to illustrating the primary use of the READNEXT statement the above program shows a method of formating a number with leading zeroes printing (see line 1120).

H.8 Example Eight - Indexed File Create

The following example illustrates a method of creating a new indexed file from a BASIC program when the programmer is unsure of the amount of contiguous disk space available. This routine allows the operator to specify the number of records desired in the file or allows the operator to specify that the file is to be allocated for the largest record count that will fit in the available space.

```
3550 C1$=AT$(1,PAGE(0))&CRT$("EOS")
3552 C2$=AT$(1,PAGE(0))&CRT$("EOS")
3554 C5$=CHR$(7)  REM Bell code
3560 REM Create new file
3561      REM S1 = record length
3562      REM S2 = file size in records
3563      REM Keylen = 30
3564      REM S is number of bytes to be used for record+key+overhead storage
3565      REM S0 is number of bytes to be used for sequential record pointers
3566      REM V1 is number of contiguous bytes available on disk
3570      OPTION CASE "U"
3580      PRINT C1$;"Please mount the disk to contain the file";F$
3590      PRINT "in the appropriate drive (Y/N)? ";R  V$="N"
3600      S1=158 \ REM S1=RECLEN
3630      LINPUT USING V$,V$
3640      IF INP=17 OR INP=26 OR V$="" OR V$="N" THEN 3940
3650      PRINT C1$;"How many records do you wish allocated? ";
3660      LINPUT USING "      ",V$
3670           IF V$="" THEN S2=999999 \ GOTO 3700
3680           IF NBR(V$)=0 THEN PRINT C5$; \ GOTO 3650 ELSE S2=VAL(V$)+3
3690           IF S2<=0 THEN S2=999999
3700      PRINT C2$;"What is the largest area on the disk? ";
3710      LINPUT USING "      ",V1$
3720           IF V1$="" OR NBR(V1$)=0 THEN PRINT C5$; \ GOTO 3700
3730           V1=VAL(V1$)*1024 \ REM Convert to bytes
3740      S3=S1+32 \ REM KEYLEN=30, overhead=2, S3=KEYLEN+RECLEN+2
3750      IF S2>V1/S3 THEN S2=INT(V1/S3) \ REM S2 must be realistic
3760      IF MOD(S2,4)<>3 THEN S2=S2-1 \ GOTO 3760
3770      IF S3*S2+S2*2>V1 THEN S2=S2-4 \ GOTO 3770\REM Make more realistic
3779      REM Take into account the overhead of rounding up to nearest 1024
3780           S=S2*S3 \ IF MOD(S,1024)>0 THEN S=S+1024-MOD(S,1024)
3790           S0=S2*2 \ IF MOD(S0,512)>0 THEN S0=S0+512-MOD(S0,512)
3800           IF S+S0>V1 THEN S2=S2-4 \ GOTO 3780\REM Make sure it will fit!
3810      REM Make S2 a prime number
3820           FOR I=S2 TO 3 STEP -4
3830                FOR J=3 TO SQR(I) STEP 2
3840                 J1=I/J
3850                 IF INT(J1)=J1 THEN 3870
3855                 NEXT
3860                GOTO 3880
3870                NEXT
3880           S2=I
3890      CLOSE #1
3900      CSI "CREATE "&F$&" (IND KEY 30 REC "&STR(S1)&" FILE "&STR$(S2)
```

Note that line 3570 will force input to be upper case only. Line 3640 validates the Y/N input--default to NO--and checks for exit (CTRL/Q or CTRL/Z indicate exit). Lines 3680 and 3720 validate the input for a numeric value.

Line 3750 forces S2 to be a value that is close to the value to be used. Line 3760 then forces S2 to be the next lowest value whose remainder is 3 when divided by 4 (a requirement for indexed file sizes). Line 3770 then forces S2 to be a value that would fit in the available space but does not take into account any rounding to the nearest 1K boundary. Lines 3780 through 3800 then adjust S2 to account for rounding, keeping modulo 4 of S2 = 3.

Lines 3810 through 3880 then force S2 to be a prime number (another requirement of indexed file sizes). The STEP value of -4 keeps the modulo 4 of S2 = 3.

APPENDIX I

ANSI MINIMAL BASIC

Functions: ABS ATN COS EXP INT LOG RND SGN SIN SQR TAN TAB

DATA
```
<line-no> DATA <literal>[,<literal>]...
```

DEF
```
<line-no> DEF FNx [(<var>)]=<exp>
```

DIM
```
<line-no> DIM <var> (<int>[,<int>])[,<var>(<int>[,<int>])]...
```

FOR
```
<line-no> FOR <var>=<exp> TO <exp> [STEP <exp>]
```

GOSUB
```
<line-no> GOSUB <line-no>
```

GOTO
```
<line-no> GOTO <line-no>
```

IF
```
<line-no> IF <rel> THEN <line-no>
```

INPUT
```
<line-no> INPUT <var>[,<var>]...
```

LET
```
<line-no> LET <var>=<exp>
```

NEXT
```
<line-no> NEXT <var>
```

ON
```
<line-no> ON <exp> GOTO <line-no>[,<line-no>]...
```

OPTION
```
<line-no> OPTION BASE 0
<line-no> OPTION BASE 1
```

PRINT
```
<line-no> PRINT [<exp>[<punct>[<exp>]]...]
```

RANDOMIZE
```
<line-no> RANDOMIZE
```

READ
```
<line-no> READ <var>[,<var>]...
```

REM
```
<line-no> REM <characters>
```

RETURN
```
<line-no> RETURN
```

STOP
```
<line-no> STOP
```

ANSI Minimal BASIC, BSR X3.60 only requires: string length of 18 characters; six significant digit accuracy; dimensioned arrays for numeric variables (not string); string variable names have <letter><$> only; single statement lines. Keywords must be separated from operands by at least one space and keywords may not be abbreviated.

If you wish to write BASIC programs that are portable from machine to machine, the above restrictions should be kept in mind to minimize the changes required.

APPENDIX J

CHARACTER CODES

Value	ASCII	Usage	Value	ASCII	Value	ASCII	Usage
0	NUL	CTRL/@	43	+	86	V	
1	SOH	CTRL/A	44	,	87	W	
2	STX	CTRL/B	45	-	88	X	
3	ETX	CTRL/C	46	.	89	Y	
4	EOT	CTRL/D	47	/ Slash	90	Z	
5	ENQ	CTRL/E	48	0	91	[	left bracket
6	ACK	CTRL/F	49	1	92	\	back slash
7	BEL	CTRL/G	50	2	93	]	right bracket
8	BS	CTRL/H	51	3	94	^	circumflex
9	HT	CTRL/I	52	4	95	_	underscore
10	LF	CTRL/J	53	5	96	`	back quote
11	VT	CTRL/K	54	6	97	a	
12	FF	CTRL/L	55	7	98	b	
13	CR	CTRL/M	56	8	99	c	
14	SO	CTRL/N	57	9	100	d	
15	SI	CTRL/O	58	:	101	e	
16	DLE	CTRL/P	59	;	102	f	
17	DC1	CTRL/Q	60	<	103	g	
18	DC2	CTRL/R	61	=	104	h	
19	DC3	CTRL/S	62	>	105	i	
20	DC4	CTRL/T	63	?	106	j	
21	NAK	CTRL/U	64	@	107	k	
22	SYN	CTRL/V	65	A	108	l	
23	ETB	CTRL/W	66	B	109	m	
24	CAN	CTRL/X	67	C	110	n	
25	EM	CTRL/Y	68	D	111	o	
26	SUB	CTRL/Z	69	E	112	p	
27	ESC	CTRL/[	70	F	113	q	
28	FS	CTRL/\	71	G	114	r	
29	GS	CTRL/]	72	H	115	s	
30	RS	CTRL/^	73	I	116	t	
31	US	CTRL/_	74	J	117	u	
32	SP	SPACE	75	K	118	v	
33	!		76	L	119	w	
34	"		77	M	120	x	
35	#		78	N	121	y	
36	$		79	O	122	z	
37	%		80	P	123	{	left brace
38	&	ampersand	81	Q	124	|	vert line
39	'	quote	82	R	125	}	right brace
40	(		83	S	126	~	tilde
41	)		84	T	127	DEL	rubout
42	*	asterisk	85	U			

A more complete character set is documented in the OASIS System Reference Manual.