AlphaBasic™

User's Manual

©-1977 ALPHA MICROSYSTEMS
ALL RIGHTS RESERVED

I would like to express my thanks to the following people
for their assistance in the development of AlphaBasic:

Mike Roach for debugging and suggestions
on implementation of new features

Paul Allen Edelstein for the trig package
and advanced mathematical assistance

Bob Hitchcock for operator and applications
oriented suggestions and scaled arithmetic

the myriad of users who pointed out problems
in the earlier versions (sometimes not so tactfully)

and most of all to Carolyn
without whom much of this may never have been accomplished.

Dick Wilcox

'AMOS', 'AlphaBasic', and 'AM-100'

are trademarks of products
and software of

ALPHA MICROSYSTEMS
Irvine, CA 92714

© 1977 - ALPHA MICROSYSTEMS

ALPHA MICRO
17881-F Sky Park North
Irvine, CA 92714

INDEX

Note: This manual reflects AlphaBasic version 3.3 release

INTRODUCTION TO ALPHABASIC

AlphaBasic is an extension of the popular BASIC language with several features not found in other implementations. These features not only enhance the performance of traditional uses of the language but also make business applications easier to program. COBOL users will find the I/O processing convenient for data manipulation while the memory mapping system will entice the assembly language programmers who wish to link up their own external routines. Floating point hardware in the processor is fully supported making AlphaBasic faster for mathematical computations than any other BASIC currently implemented in a microprocessor system.

AlphaBasic runs in one of two modes. Interactive mode operates in the traditional manner whereby the user creates, alters, and tests his program which resides totally in memory. This mode is convenient for the creation and debugging of new programs or the dynamic alteration of exisitng programs. Compiler mode is more useful for programs which are to be put into productive use or for testing programs which are too large to fit in memory in the interactive mode. In compiler mode, the user compiles the program and stores the compiled object code on the disk. During the actual running of the compiled program, only the object code and a minimal run-time execution package need be in memory thereby conserving space and increasing run speed.

The compiler and the runtime package are both written in reentrant code so that they may optionally be shared by all users running or debugging programs. The object programs created by the compiler are also totally reentrant and sharable thereby further reducing memory requirements if it is desired to allow several users to run the same program.

Data formats supported include floating point, string, binary and unformatted variables. All data formats may be simple variables or array structures. In addition, the unique memory mapping system allows the user to specify the ordering of variables in prearranged groupings for more efficient processing. This system is similar to the data formatting capabilities of the COBOL language and lends itself nicely to business applications where the manipulation of formatted data structures is of prime concern. Advanced compiler techniques have been used in all areas to give a truly commercial grade processing system which may be easily integrated into a series of programs not limited to the BASIC language. If the user is running a large number of BASIC tasks in a timesharing environment, the run-time package is fully reentrant and may be included in the resident monitor for a further increase in memory efficiency. The compiled object programs are also totally reentrant and sharable for users who are running the same application programs.

Variable names are not limited to the conventional single character and single digit format but may be any number of alphanumeric characters in length, as long as the first character is alphabetic. This is accomplished by using a dynamic tree structure for the storage of reserved words and variable names within the compiler and is another feature which makes AlphaBasic ideally suited for business applications. Since the source code is compiled and need not be in memory when the program is eventually run, the length of the variable name is not a significant concern. Label names may also be used to identify points in the program for GOTO and GOSUB branches. Label names are alphanumeric and help

to clarify the program structure when used with meaningful definitions.

The following sections will describe AlphaBasic features and operations. It is assumed that the reader is familiar with conventional BASIC language concepts. This initial description is not meant as a tutorial to the novice but rather as an informational packet which will list the supported functions of conventional BASIC and give more detailed study only to those areas that differ from the norm. In all cases, AlphaBasic has been designed to support all features of conventional BASIC which is currently in use in the microprocessor field. In those few areas where no mention is made here of a feature which is normally considered standard, it is probably due to lack of space (or possibly lack of time) during the initial writing of this description. The reader is encouraged to contact his local dealer to clear up any specific questions that may arise until such time as more comprehensive documentation is generated on AlphaBasic.

This section will attempt to explain the general differences encountered in the AlphaBasic compiler system and to list the currently implemented features or problems and limitations that are known to exist at this time.

COMPILER OPERATION

The user initiates the AlphaBasic compiler function by entering the command BASIC while in monitor command mode. Once the compiler has been located and loaded into RAM it will print the READY message and await user input. Although the system is a compiler in actual operation it has been designed to look as much like the popular interpreters that are currently on the market. The system is interactive in nature and the user may enter his source program and edit it on a line number basis just like the interpretive versions. No fancy editing techniques are yet available and each line must be changed by entering the line number and the entire new line. Line numbers must be in the range of 1-65534 to be valid. The source text is built up in memory as it is entered and is automatically kept in numerical sequence. Multiple lines (terminated by line-feed instead of carriage-return) are not supported at this time but will be in the future. No syntax checking is performed when the line is entered other than validation of the line number.

When the user enteres the RUN command the source program is compiled in its entirety and syntax errors reported. The resulting object code is also stored in memory which results in a much greater initial memory requirement. If no errors were detected the object code is then executed with no direct reference to the source code anymore. Any further editing of the source code will set a switch to automatically force recompilation when the next RUN command is entered.

The user may store the compiled object program on disk by using the SAVE command with the explicit extension "RUN" appended to the program name. This saved program may then later be executed without recompilation by use of the monitor RUN command which calls upon the runtime package only.

MULTIPLE STATEMENT LINES

The system supports multiple statement lines by using colons to separate the statements. The normal rules apply such as REM and DATA statements cannot contain other statements on the same line. Immediate mode commands may also be multiple statement lines.

IMMEDIATE MODE COMMANDS

All lines typed into AlphaBasic by the user will be considered for immediate execution if no line number preceeds the command. System commands result in immediate interpretation and execution and are considered those commands which result in a system fucntion but are not ever included in the text of a program itself. Normal program statements may be entered without a line number in which

case they will be compiled as a single line program and then immediately executed. Certain commands are considered illegal in immediate mode and the user is advised that no error messages are currently implemented to prevent the user from inadvertantly attempting to execute these commands. In some instances the execution of these commands will result in complete system destruction which means you must reboot the entire operating system. This malady will be corrected in a future release.

VARIABLE NAMES

Variable names are not limited to a single letter or a letter and a digit as in conventional BASIC implementations. A variable name may contain any number of alphanumeric characters as long as the first one is alpha A-Z. Apostrophes may also be used in variable names to improve clarity. Mapped variables are defined by an explicit type code and therefore do not follow the standard convention of using a dollar sign for string variables. Normal (non-mapped) variables are considered floating point variables unless they are terminated by a dollar sign in which case they are considered string variables. Subscripting follows the standard conventions of other BASIC's by enclosing the subscripts within parenthesis. Some examples of legal variables follow:

```
A
A$
NUMBER
STRING$
MASTER'INVENTORY'RECORD
HEADER1
MOM'ALWAYS'LIKED'YOU'BEST
Z1234567
```

PROGRAM LABELS

AlphaBasic allows the use of program labels to identify points in the program. A program label is composed of 1 or more alphanumeric characters of which the first must be alpha A-Z. Apostrophes may also be used within labels for clarity. Labels, when used, must be the first itme on a line and must be terminated by a colon (:). A label may be followed by a program statement on the same line or it may be the only item on the line. Labels operate similar to line numbers for GOTO and GOSUB statements and make the program easier to document. An example of label usage follows:

```
10 START'PROGRAM:
20 INPUT "ENTER TWO NUMBERS TO GET SUM: ",A,B
30 PRINT A;"+";B;"="A+B
40 IF A+B<>0 GOTO SUM'NOT'ZERO
50 PRINT "SUM IS ZERO"
60 GOTO START'PROGRAM
70 SUM'NOT'ZERO: PRINT "SUM IS NOT ZERO"
80 GOTO START'PROGRAM
90 END
```

MEMORY ALLOCATION

The compiler system allocates memory dynamically during editing, compiling, and execution of the user program. Currently there is no check made to insure that the user is not running out of memory in his allocated partition and chances are pretty good that memory overrun will result in a system crash. We realize that this is a grave restriction and there are several technical reasons why this condition is not checked for and reported as an error. I will not bore you with the gory details on this except to insure you that it will be implemented in a future production relase of AlphaBasic. In the meantime you can kinda keep tabs on memory usage with the MEM(X) function which will deliver back various usages of memory during program development. See the section on functions for more detail on this.

EXPANDED SOURCE TEXT MODE

AlphaBasic normally scans the source text in expanded mode (version 3.2) which dictates that reserved words (verbs, functions, commands, etc) be terminated by a space or a character that is illegal in variable names. This allows labels and variables to begin with reserved words. In other words, the variable name PRINTMASTER will not be interpreted as PRINT MASTER in expanded mode. In this mode the statement FOR A=1 TO 10 cannot be written as FORA=1TO10. There are two commands which the user may apply to switch back and forth between normal and expanded modes:

```
EXPAND          !sets syntax scanner to expanded mode
NOEXPAND        !sets syntax scanner to normal mode
```

AlphaBasic initializes itself in the expanded mode. Note that the mode in which a program is compiled has nothing to do whatsoever with the resultant object code which is generated either in size or execution speed.

INTERACTIVE VS COMPILER MODES

AlphaBasic may be run in one of two modes of execution. Interactive mode deals directly with the user on a one-for-one basis allowing direct editing of the source program in memory and testing of that program in a debug mode. This is the mode that most BASIC interpreters operate in and the one that most users of BASIC are already familiar with. In addition to the normal commands there are some unique features that make AlphaBasic more useful in a debug mode.

In interactive mode the entire compiler as well as the user source program must be stored in memory for operation. Editing of the source program takes place in the conventional manner by typing each line with its line number first. Lines are kept in sequence automatically by the internal editing routines. Each time a change is made in the source program a switch is set which indicates that the program must be recompiled before it can be executed. Any time the user enters a command which results in program execution (RUN, CONT, or single-step) this switch is tested and if set, the program is compiled in memory and the object code generated in its own special area of memory also. It is this object code that is then executed by the run-time package which also must be in memory during interactive operations.

Commands which do not have a line number are considered immediate mode commands and are executed immediately. Actually, the statement is compiled and execution is applied against the currently existing set of defined variables. New variables are defined as required by immediate mode commands. Certain commands are meaningless in immediate mode and are therefore not allowed (FOR, NEXT, etc). Multi-statement lines are lines which contain more than one command separated by colons and are allowed in immediate mode as well as in the source program.

A unique feature which is very useful for debugging is the single-step command. Each time the line-feed key is entered with no data on the line, the next instruction in the program is listed and then executed. Control is then returned to the user terminal for inspection of variables or alteration of the source program before continuing execution. Note that any alteration of the source program results in recompilation before the single-step command is actually executed.

Compiler mode differs from interactive mode in that the program is compiled first and then the object code is stored on disk with a special save command. The object code may then be run at any time in the future requiring that only the run time package be in memory along with the object code itself. Neither the compiler proper nor the program source code is required for execution in compiler mode resulting in a large cutback in memory requirement. This mode is also ideal for sequential automatic processing of multiple programs within a system structure. IN this mode, BASIC programs may be freely mixed with assembly language programs for maximum efficiency and flexibility.

RUNNING BASIC PROGRAMS

AlphaBasic supports the ability to compile and run programs without having to reread the source program by saving the compiled code on disk and using the runtime package to run that code at any later time. There are two main components in the compiler system named BASIC.PRG and RUN.PRG. These programs reside in the system library (DSK0:[1,4]) and are called into memory as they are required. BASIC.PRG is the compiler and contains the code which scans the source program in memory and creates the runnable object program also in memory. All system commands such as LIST, LOAD, SAVE, etc. are also processed in this program. RUN.PRG is the runtime package and contains the code and subroutines which perform the actual execution of the compiled object code from memory. This object code can be from the compilation phase during interactive program development or it can be from the direct loading of a saved object code file on the disk.

The runtime package (RUN.PRG) is free-standing and may run without referencing the compiler program (BASIC.PRG). The compiler, however, will first insure that the runtime package is present in memory and load it if necessary since it draws upon several of the internal runtime routines during compilation. The user may optionally build his monitor with the runtime package resident for sharing by all users. Those users which then run in the interactive mode will also benefit since BASIC.PRG will locate RUN.PRG in the monitor memory and not have to load it again into the user memory. The compiler itself may also be included in the monitor but due to the size of it this is not normally done unless the majority of the users will be doing program development and testing (such as a school training program teaching BASIC).

To run in the interactive mode (the mode most familiar to us due to other versions of BASIC) the user types the command "BASIC" on his terminal and the compiler and runtime programs are both located and loaded into memory as required. The LOAD command may then be used to load source programs into memory for editing, compilation and test. The SAVE command may be used to save either the source program or the compiled object program on disk for later use. Source programs have the extension "BAS" while compiled object programs have the extension "RUN". Note that there is no way to recover the source text from the compiled object code so the normal rule is to save both of them either on the same or different disks. Running in the interactive mode always involves the compilation and running of a source program which is in memory and never includes running a saved object code program directly.

To run a compiled program the user must be in monitor mode. Monitor mode may be entered from the interactive compiler mode above by exiting the compiler via the BYE command typed on the terminal. Once the monitor mode is obtained the user enters the RUN command followed by the name of the object program to be run. The runtime package is located (and loaded from disk if necessary) and then started. The runtime package then initializes user memory and locates the user program specified in the RUN command. This program has the extension "RUN" and may be either in memory already or be loaded from the user disk area. Loading of the program is automatic by the runtime package if the program is not in memory. The program is then executed and continues until the end or until aborted by the user typing a control-c on his terminal.

Note that the RUN command serves two different functions depending on whether the user is in monitor mode or in interactive BASIC mode. In monitor mode the RUN command is used to execute a compiled BASIC program which has been previously stored on disk or loaded into memory explicitly. The command RUN PAYROL will run the PAYROL.RUN compiled program and then exit back to monitor mode without ever entering the compiler itself. In interactive BASIC mode the RUN command will compile and run the current source program which the user is editing and testing. The user should note these differences.

While we are on the subject of running source programs versus compiled object programs, there is one more restriction that the user should note. The CHAIN statement causes one program to terminate execution and turn control over to the next program in sequence. This chained program must be a compiled object program or it will not be found. There is no mechanism to load a source program and then compile it for the CHAIN statement's use. The object program module may, however, be already in memory or it may be on disk in which case it will be loaded automatically. If the chained program is already in memory there will be no wait time due to loading from the disk. The object program module may be loaded into memory by the monitor LOAD command (different from the AlphaBasic LOAD command).

Note also that the compiled object program is totally reentrant and sharable between several users so that any programs that are commonly used as either complete programs or chained links may be stored in monitor memory for high-speed access.

Variable names may be any number of unique alphanumeric characters and are not limited to a single character or character-digit pair as in other versions of BASIC. The first character of the name must be alphabetic and the variable name must not begin with any reserved word used in BASIC. Apostrophes may also be used in variable names to improve clarity. Mapped variables may take on any type format regardless of the name terminator. Unmapped variables assume the type code as in other versions of BASIC. String variables are specified by appending a dollar sign to the name and integer variables are specified by appending a percent sign to the name. Refer to the section on data formats for more detail on these options.

NUMERIC VARIABLES

The normal mode of processing mathematical variables (as opposed to string variables) is in 11-digit accuracy which might be termed "single-and-one-half" precision compared to normally accepted standards. This is due to the hardware floating point instructions which are implemented in the WD-16 microprocessor chip set used in the AM-100 computer. Integer and binary variables are also considered numeric variables but are always converted to floating point format prior to performing mathematical operations on them. All printing of math variables is done under normal BASIC format with the significance being variable under user control from 1 to 11 digits. The SIGNIFICANCE statement is used to set up this value.

STRING VARIABLES

AlphaBasic supports string variables in both single and array modes. The memory that is allocated for each string variable is the number of bytes representing the maximum size that the string is allowed to expand to. Each string is variable in size within this maximum limit and a null byte is stored at the end of each string to indicate its current actual size. At the start of each compilation the default size to be used for strings is 10 characters maximum. The STRSIZ statement may be used within the program to alter the value to be used for all new string variables which follow. Future releases of the compiler syntax will allow individual string definitions to have their own size specification without the need to change this default value.

String variables are manipulated in the same manner as in other versions of BASIC and may be concatenated by use of the plus sign between two strings. String variables may be assigned values by enclosing string literals in quotes. String functions such as LEFT$, RIGHT$, MID$, etc. are implemented to assist in manipulating portions of strings or substrings. In addition, a powerful substring modifier may be used to operate on portions of strings within expressions. A seperate section is devoted to this unique option of AlphaBasic.

Unformatted variables are also considered string variables when they are used in expressions or printed. Be careful with this one! If the unformatted variable is mapped to contain subfields which are not in string format, it will cause

some very strange results when printed or used in expressions. This is because no conversions are performed on the subfields of an unformatted variable; it is used in its entirety exactly as it appears in memory.

VARIABLE ARRAYS

Arrays may be numeric or string variables and are allocated dynamically during execution when the DIM statement is encountered in the program. During execution if no DIM statement has been encountered when the first reference to the array is made, a default array size of 10 elements for each subscript level is used. This means that all DIM statements must be executed in the program prior to any actual references to the array.

Arrays may be any number of levels deep but practicality dictates some reasonable limit of 20 or so. Each level is referenced by a subscript value starting with element 1 and extending to element N. Once an array has been dimensioned by a DIM statement it may not be redimensioned by a subsequent DIM statement in the same program. At no time may the number of subscripts vary in any of the references to any element in the array. The number of subscripts in each element reference must also match the number of subscripts in the corresponding DIM statement which defined the array size.

AlphaBasic has been designed with the goal of flexibility in interfacing with other language processors within the AMOS operating system and primarily aimed towards assembly language programs where the data from a BASIC program must be manipulated in a way that is either infeasible or inefficient for another BASIC program to do. In order to accomplish this goal, the data formats must be clearly defined and understandable by the user who wishes this information. Data will normally be output to one or more disk files to be passed on to another BASIC program or to an assembly language program for further reduction or processing. Data may also be output in a print image format for printing by the operating system print spooler job. It is our intention to provide all required information on the internal formats and workings of AlphaBasic so that the user will find the task of interfacing the software outputs to be straightforward. This initial description is not meant to be the final result of the internal documentation effort, but merely an introduction to the basic design theory behind the compiler.

All variables in use by the application program are stored in a dynamically alterable area within the user program area. The areas that change during program execution are those which are set aside for arrays and variable size strings. Simple variables do not change position once they are assigned storage positions. Assignment of storage is normally done as each variable is encountered in the source program. The user has the option of overiding this assignment by the memory mapping system which will be described later. Each variable is assigned as one of the following data format types and once assigned, may not be changed within the same program.

FLOATING POINT - all numeric variables are assigned as this type unless otherwise specified in the program. The standard precision in use by this system would probably be called "single-and-one-half" since it lies midway between what has been accepted as single and double precision formats. The reason for this is that the hardware floating point instructions all work in this format and so we may as well make use of the extra precision wherever possible. Floating point numbers occupy six bytes of storage and are in the same format as dictated by the hardware instructions FADD, FSUB, FMUL, FDIV, and FCMP. Of the 48 bits in use for each 6-byte variable, the high order bit is the sign for the mantissa. The next 8 bits represent the signed exponent in excess-128 notation giving a range of approximately 2.9*10-39 thru 1.7*10^38. The remaining 39 bits contain the mantissa which is normalized with an implied high-order bit of one. This gives an effective 40-bit mantissa which results in an accuracy of 11 significant digits.

STRING - used for the storage of alphanumeric text data. String variables may be assigned fixed lengths for efficiency and speed or may be left to dynamically vary in size as the data changes. Fixed length string variables require one byte of storage for each character and may be fixed in position using the memory mapping system. Dynamic length string variables may not be mapped into memory since their position may shift during execution. An indexing scheme is defined to the user if it is absolutely necessary for him to locate these strings with an external routine.

BINARY - binary variables are similar to integer variables in other implementations of BASIC. A binary variable may be from 1 to 5 bytes in length and may be signed when all 5 bytes are specified. When less than 5 bytes are specified for the size (in a MAP statement) the binary value may be loaded as a negative number but will always be returned as a positive number of full magnitude with the upper bit (preloaded as the sign) taking on its specific value in the equivalent positive binary variable. For instance, a 1-byte binary may be loaded with positive numbers from 0 thru 255 (decimal) or negative numbers from -1 thru -128 but the negative numbers will be returned as the positive values of 128 thru 255 respectively. Only 5-byte binary variables will return the original sign and value when loaded with a negative number.

Binary variables may be used in expressions but they are slower than floating point variables because they are always converted first to floating point format before any mathematical operations are performed on them. Binary variables are useful in integer and logical (Boolean) operations or for storing values in small amounts of memory (floating point numbers always take 6 bytes of memory regardless of their values). All logical operations performed within expressions (AND, OR, XOR, NOT etc.) cause the values to be first converted to signed 5-byte binary format before the logical operation is performed. The value -1 represents a 40-bit mask of all ones and it is this value which is returned as the result of any relational comparison between two expressions or variables.

INTEGER - integer variables and constants are specified by appending the variable name with a percent sign (%) which is the standard convention in use by other BASIC's. AlphaBasic generates floating point variables and performs automatic integer truncation for all integer variables specified in this manner. Integer constants are generated as their equivalent floating point values and are included only for compatibility with existing program structures. Since integer variables are effectively floating point variables with an additional INT conversion performed, they are actually slower than regular floating point variables. This is opposite from most other BASIC's which usually store integer variables as 2-byte signed values and perform special integer arithmetic on them. True integer variables may be defined by using the MAP statement and the "B" binary type code.

UNFORMATTED - defines a fixed size area of storage used to contain absolute unformatted data which may be in any of the above formats. This format will normally be used in the mapping system to define contiguous storage which is subdivided into multiple variables of different formats. No conversion ever takes place when moving data to and from this format. Unformatted variables are treated as string variables when used in expressions.

* Note that dynamic length string variables are not yet implemented.

Expressions in AlphaBasic follow the same format used by other popular versions of BASIC. Parentheses are used to designate hierarchy within expression terms and the normal mathematical hierarchy prevails in the absence of parentheses. The following mathematical operators are recognized by AlphaBasic:

+	unary plus or addition
-	unary minus or subtraction
*	multiplication
/	division
^	raise to power
**	raise to power
"	string literal
NOT	logical not
AND	logical and
OR	logical or
XOR	logical xor
EQV	logical equivalence
MIN	minimum value
MAX	maximum value
=	equal
<	less than
>	greater than
<>	unequal
><	unequal
#	unequal
<=	less than or equal
=<	less than or equal
>=	greater than or equal
=>	greater than or equal

OPERATOR PRECEDENCE

The precedence of operators determines the sequence in which mathematical operations are performed when evaluating an expression that does not have overriding parenthesis to dictate hierarchies. AlphaBasic uses the following operator precedence:

exponentiation
unary plus and minus
multiplication and division
addition and subtraction
relational operations (comparisons)
logical AND, OR, XOR, EQV, MIN, MAX
logical NOT

MODE INDEPENDENCE

Expressions may contain any mixture of variable types and constants in any arrangement. AlphaBasic performs automatic string and numeric conversions as

necessary to insure the result is in the proper format. For example, if two strings are multiplied together they will first be automatically converted to numeric format before the multiplication takes place. If the result is then to become a string it will be reconverted back to string format before the assignment is performed. In other words, the statement A$ = B$ * "345" is perfectly legal and will work correctly. This is a powerful feature which can save much programming effort when used correctly.

There is an ambiguous situation which arises from this mode independence. The plus symbol (+) is used both as an addition operator for numeric operations and as a concatenation operator for string operations. The value of 34+5 is equal to 39 but the value of "34"+"5" is equal to the string "345". The operation of the plus symbol is unambiguaous in its operation but may take a little thought to figure out its exact usage in all situations. A few examples might help.

If the first operand is numeric and the second is string we convert the second to numeric and perform addition.

34 + "5" equals 39

If the first operand is string and the second operand is numeric we convert the second to string and perform concatenation.

"34" + 5 equals "345"

The above two examples apply only when we are not "expecting" a particular type of variable or term. This generally occurs only in a first level PRINT expression. At other times we are expecting a specific type of variable and the conversion of the first variable will be performed prior to inspecting the operator (plus sign). The operation of the plus sign is then implicitly specified by the result of the first first variable. Take the following example:

5 * "34" + 4

The multiplication operator (*) forces us to expect a numeric term to follow. The "34" string is therefore immediately converted to numeric 34 and multiplied by the 5. The plus sign then performs numeric addition instead of concatenation. The result is in numeric format and will be converted if its destination is a string.

If this approach seems confusing you should try a few examples of your own on the system to see what the results are. Remember, any potentially ambiguous expression may always be forced to one or the other type by use of the STR and VAL functions.

LOWER CASE CHARACTERS

Beginning with release 3.1 AlphaBasic supports lower case letters (a-z) in both the input source program and in the runtime execution of programs. The line editor built into the interactive system will now accept and store source input text in lower case characters if desired. Lower case letters when used within variable names and labels will be unique and separate from the corresponding upper case letters. In other words, the variable "a" is separate from the variable "A" and the variable "Tom" is separate from the variables "TOM" and "tom". Lower case letters may be used as the first character of a variable name or program label just as upper case letters may be.

Reserved words are treated somewhat differently from the above system. When a reserved word is expected, the syntax parser temporarily translates all lower case letters to upper case and then checks for a reserved word match. If the word is not a reserved word the translation is not retained and the lower case letters are used for variable name matches. The following statements are all considered to be identical:

```
FOR A = 1 TO 100 STEP 2
For A = 1 To 100 Step 2
For A = 1 to 100 step 2
for A = 1 to 100 step 2
```

Lower case letters used within string literals (inside quotes) will be retained and printed as lower case. Lower case letters which are entered into string variables by means of the INPUT statement will also be retained as lower case letters. The entire string processing system now supports lower case characters.

Note that all lower case characters are considered greater than any upper case character due to their position in the ASCII colating sequence. To assist in processing and comparing input which contains lower case letters the UCS(X) function has been implemented. This function will return a string which is identical to the argument string (X) with all lower case characters being translated to upper case. The inverse function LCS(X) will return a string with all upper case characters being translated to lower case.

SUBSTRING MODIFIERS

AlphaBasic supports a unique method of manipulating substrings. A substring is defined as a portion of an existing string which may be as small as a single character or as large as the entire string. Substring modifiers allow the substring to be defined in terms of character positions within the string, relative to either the left or right end of the string. The length of the substring is defined either in terms of its beginning and ending positions or in terms of its beginning position and its length.

Substrings are defined by referencing the desired string followed by the substring modifier. The substring modifier is two numeric arguments enclosed within square brackets. The substring modifier takes on two distinct formats:

```
[beginning-position,ending-position]
[beginning-position;substring-length]
```

The first format defines the substring in terms of its beginning and ending positions within the string and uses a comma to separate the two arguments. The second format defines the substring in terms of its beginning position within the string and its length using a semicolon to separate the arguments. The second format basically performs the same function as the MID$ function.

The beginning and ending positions may are defined as character positions within the string relative to either the left or right end. A positive value represents the character position relative to the left end of the string with character position 1 representing the first (leftmost) position. A negative value represents the character position relative to the right end of the string with character position -1 representing the last (rightmost) position. Assume the following string has the letters ABCDEF in it. The positions are defined in terms of positions 1 through 6 (left-relative) or positions -1 through -6 (right-relative).

```
 A  B  C  D  E  F    (6 characters within main string)
 1  2  3  4  5  6    (left-relative position values)
-6 -5 -4 -3 -2 -1    (right-relative position values)
```

Allowing negative values for right-relative positions provides the ability to pick out digits within a numeric string without having to calculate the total size of the string first and then work from the left.

The substring-length argument used by the second format may also take on negative values for a more flexible format. Normally the length is a positive value which represents the number of characters counting the beginning position and incrementing the index to the left. A negative length causes the index to move to the right and returns a substring whose last character is the one marked by the beginning-position argument. Confusing? Perhaps a few examples may clarify the use of substring modifiers. Assume the main string is A$ and it contains the above example of "ABCDEF". The following substrings will be returned:

```
A$[2,4]     equals    BCD
A$[2;4]     equals    BCDE
```

```
A$[3,3]      equals    C
A$[3;3]      equals    CDE
A$[-3,-2]    equals    DE
A$[3,-2]     equals    CDE
A$[3;-2]     equals    BC
A$[-3;-2]    equals    CD
A$[4;1]      equals    D
A$[4;-1]     equals    D
```

Any position values or length values which would cause the substring to overflow out of either end of the main string will be truncated at the string end.

```
A$[3,10]     equals    CDEF
A$[-14,34]   equals    ABCDEF
```

The main string to which the substring modifier is applied is actually any expression and need not be a defined single string variable.

```
Q$ = (A$+B$+C$)[2;10]
Q$ = ("ABLE"+A$+"QQ34")[4,10]
```

The mode independence feature allows substring modifiers to be applied to numeric expressions. A string is returned but if the destination is a numeric variable another conversion will be made on the substring to return a numeric value.

```
Q = (A*B)[2,5]
PRINT X[3,4]
```

Substring modifiers may be applied to subscripted variables or expressions containing subscripted variables. The substring modifiers are independent functions not to be confused with subscripts.

```
Q$ = A$(3,4)[2,5]
Q$ = (A$(1)+B$(3))[-5,3]
```

Substring modifiers return a string value. These may be used the same as strings in expressions.

```
Q$ = A$ + B$[2;5] + (A$[2,2] + C$)[-5;-3]
```

Substring modifiers may be applied to the left side of an assignment in order to alter a substring within a string variable. Only that portion of the string defined by the substring modifier will be changed. The other characters in the string will not be altered. This may not be applied to numeric variables.

```
A$[2,4] = "QRS"
```

In the above example, if A$ had contained "ABCDEF" before the assignment was executed the result in A$ would be "AQRSEF".

AlphaBasic derives much of its versatility and speed from the fact that it is a true compiler in its execution mode. Memory storage is allocated during compilation for all defined variables in an area that is contiguous and predictable. All variables are referenced by the compiled program code through an indexing scheme. Each variable in the working storage area contains an item in the index area which contains all information needed to define and locate that variable. The working storage area therefore contains only the pure variables themselves without any associated or intervening descriptive information. The index area is a separate entity physically located before the working storage area in memory.

The allocation of the variable storage area for any program is predictable and normally done as each variable is encountered during compilation. Since this scheme is not easily followed by the user, a different method must be derived which can override normal allocation processes for those users who wish to have the variables allocated in a predetermined manner. The disk I/O system also requires that the variables used be in a specific relationship to each other when used in the more advanced modes. The MAP statement has been included in AlphaBasic for the purpose of allocating variables in a specific manner. MAP statements are non-executable at run time but merely direct the compiler in the definition and allocation of the referenced variables. Each MAP statement contains a unique variable name to which the statement applies. When the compiler encounters this statement it allocates the next contiguous space in working storage as required and assigns it to that variable name. The type of the variable is also specified in this statement and may be used to override the standard naming conventions of BASIC. All variables not defined in a MAP statement will then automatically be assigned storage in sequence for total compatibility with existing standards.

As has been noted previously, this memory mapping system is primarily required for advanced disk I/O techniques and to assist in linking with assembly language routines. Special functions are provided to deliver the absolute addresses of these areas as parameters during assembly routine calls. By knowing the layout of the variables in memory, the user need only pass the base address of the area to the routine and the routine can then reference all needed variables by indexing techniques. The mapping system does have another distinct advantage to the sophisticated programmer in the allocation of arrays. With the MAP statement the user has the ability to override the standard array allocation scheme and force the allocation to proceed in a more flexible manner. Conventional BASIC arrays are allocated in contiguous memory for each subscripted variable encountered. AlphaBasic allows several variables to be combined in a single contiguous array which can provide efficiency in the manipulation of associated data structures.

MAP STATEMENT FORMAT

Briefly, the MAP statement has the following syntax:

```
MAPn variable-name(dimensions), type, size, value, origin
```

MAPn represents the hierarchy of the mapped variable for nesting purposes within other variables of a higher level. It must be within the range of MAP1 through MAP16. As each mapped variable is assigned it will be automatically included as a part of those variables with a higher level. Level numbers are actually backward with MAP1 being the highest level and MAP16 being the lowest (or innermost) level in the nesting scheme. Levels need not be sequential as they are assigned. That is, a MAP5 statement may immediately follow a MAP2 statement without having dummy MAP3 and MAP4 statements intervening if desired. This nesting scheme closely parallels the data division format in the COBOL language.

To eliminate potential allocation problems all MAP1 level variables will be forced to begin on an even address. This allows insuring that certain binary and floating point variables will begin on word boundaries if desired for assembly language subroutine processing. The AM-100 instruction set performs most efficiently when word data is aligned on word boundaries. Also, floating point variables will always be aligned to word boundaries.

VARIABLE NAME

The variable-name is the name to be given the variable for referencing within the program and must follow the rules for AlphaBasic variables. Since the type may be explicitly specified, the user need not follow the normal conventions such as requiring that the string variable names be followed by a dollar-sign for identification. If the variable name is followed by a set of subscripts within parenthesis then the variable will be assigned as an array with the dimensions specified by the subscripts just as if a DIMENSION statement had been used to assign the array. For example, the statement MAP1 A,F assigns a single floating point variable called "A" but the statement MAP1 A(5,10),F assigns a floating point array with 50 elements in it (5 X 10) just as if the statement DIM A(5,10) had been executed. Note that since these mapped arrays are assigned memory at compile time and not at run time, the subscripts must be decimal numbers instead of variables.

TYPE CODE

The type code is a single character code which specifies the type of variable to be mapped into memory. The following variable types are implemented in AlphaBasic:

X - unformatted absolute data variable
S - string variable
F - floating point variable
B - binary unsigned numeric variable

If no explicit type code is entered, unformatted data (type X) is assumed.

Unformatted data is absolute in memory and is usually only used to reference a group of other variables as one unit. Until more specific details on data handling are determined, unformatted data variables should only be moved to other unformatted data variables. For all practical purposes, unformatted data variables are treated like string variables except that they are not terminated by a null byte, only by the explicit size of the variable.

String variables are terminated either by the explicit size of the variable or by a null byte (0) if the string is shorter than the allocated size. Moving a long string to a short one truncates all characters which will not fit into the new string variable. Moving a short string to a long one causes the remainder of the long variable to be filled with null (0) bytes so that the actual data size of the string will be preserved for concatenation and printing purposes.

Floating point variables are the normal numeric variables used for mathematical calculations. AlphaBasic supports only one-and-one-half precision floating point variables which require 6 bytes of memory each and give 11-12 digits of precision. Floating point variables must begin on an even address and the MAP statement processor will leave a blank byte between any floating point variable and the variable it follows in memory if that variable ended on an odd address. This probably would only cause concern during the mapping of record structures for I/O transfers.

Binary variables may range in size from 1-5 bytes giving from 8-39 bits of binary unsigned numeric data or 40 bits of binary signed data. This is handy for the storage of small integer data such as flags in a single byte or for the storage of memory references as word values with a range of up to 65535 in two bytes. Binary variables are manipulated just like floating point variables with conversions to and from the floating point format being automatic for calculations. Since all binary variables are converted to floating point format before performing any arithmetic calculations, binary arithmetic is actually slower than normal floating point arithmetic and is used mainly for compacting data into files and arrays where the floating point size of 6 bytes is inefficient. When conversions from floating point to binary are done, any data that will not fit within the defined size of the target variable will merely be lost with no error message given. Range checks, where required, are the responsibility of the programmer prior to moving a floating point number to a binary variable area. The best way to understand this is to play with a few examples in immediate mode.

Please take note that the use of binary numeric variables is not allowed in some instances. FOR-NEXT loops may not use a binary variable as the target variable although they may be used in the FROM, TO and STEP value expressions. The record number key in a random mode OPEN statement must be floating point also. The result variable of a LOOKUP statement must be floating point.

SIZE

The size parameter in the MAP statement is optional but if used, it must be a decimal number specifying the number of bytes in the variable. If it is omitted it will default to 0 for unformatted and string types, 6 for floating point types, and 2 for binary types. The size parameter of floating point variables must be 6 or omitted.

VALUE

An initial value may be given to any mapped variable by including any valid expression in the value parameter. This value may be a numeric constant, a string constant, or a complete expression including variables. Remember,

however, that the expression is resolved when the MAP statement is executed at runtime and the current value of any variable within the value expression is the one used to calculate the assignment result. MAP statements may be executed more than once if it is desired to reload the initial values.

Note that if the size parameter is omitted (such as for floating point variables) but the value parameter is used there must be an extra comma to indicate the missing size parameter.

```
MAP1 PI,F,,3.14159
MAP1 HOLIDAY,S,10,"CHRISTMAS"
```

The first example preloads the value 3.14159 into the floating point variable called PI. The second example preloads the letters CHRISTMAS into the string variable called HOLIDAY.

ORIGIN

In some instances it may be desirable to redefine records or array areas in different formats so that they occupy the same memory area. For instance, a file may contain several different record formats with the first byte of the record containing a type code for that record format. The origin parameter will allow you to redefine the record area in the different formats to be expected. When the record is read into the area the type code in the first byte can be used to execute the proper routine for the record type. Each different routine can access the record in a different format by the different variable names in that format. All record formats actually occupy the same area in memory. This feature directly parallels the "redefine" verb in the COBOL language data division.

Normally, a MAP statement causes allocation of memory to begin at the point where the last variable with the same level number left off. The origin parameter allows this to be modified so that allocation will begin back at the base of some previously defined variable and therefore overlay the same memory area. If the new variable is smaller than the previous one (or the exact same size) it will be totally contained within the previous one. f it is larger than the previous one it will spill over into newly allocated memory or possibly into another variable area of the same level depending on whether there were more variables following it. (Play with this one a while to get the hang of it).

The origin parameter must be the last parameter on the line and takes the format @TAG where TAG is a previously defined mapped variable on the same map level. If size and value parameters are not included in this statement they may be omitted with no dummy commas if desired.

The following statements define three areas which all occupy the same 48-byte memory area but which may be referenced in three different ways:

```
100 MAP1 ARRAY
110     MAP2 INDEX(8),F
200 MAP1 ADDRESS,@ARRAY
210     MAP2 STREET,S,24
220     MAP2 CITY,S,14
```

```
230     MAP2 STATE,S,4
300 MAP1 DOUBLE'ARRAY,@ARRAY
310     MAP2 UNIT(6)
320        MAP3 CODE,B,2
330        MAP3 RESULT,F
```

Statements 100-100 define an array with 8 floating point elements for a total of 48 bytes in memory. Statements 200-230 define an area with three string variables in it for a total of 42 bytes. Normally this area would follow the 48-byte ARRAY area in memory but the origin parameter in statement 200 causes it to overlay the first 42 bytes of the ARRAY area instead. Statements 300-330 define another array area of a different format with 6 elements, each element being composed of one 2-byte binary variable (CODE) and one floating point variable (RESULT). The origin parameter in statement 300 also causes this area to overlay the ARRAY area exactly.

Caution: The above scheme allow variables to be referenced in a different format than when they were entered into memory. If you load the 8 elements INDEX(1) through INDEX(8) with floating point values and then reference the variable STREET as a string you will get the first four floating point variables, INDEX(1) through INDEX(4), which will look very strange in string format!

EXAMPLES

The following two statements produce identical arrays:

```
100 DIM A1(10)
110 MAP1 A1(10),F
```

Both statements produce arrays containing ten floating point variables referenced as A1(1) thru A1(10). Statement 110, however, will define its placement in memory in relation to other mapped variables. Similarly, the two statements at 300 and 310 produce the same two-dimensional array as the statement at 200:

```
200 DIM B1(5,20)
300 MAP1 BX(5)
310     MAP2 B1(20),F
```

Inspect the following statements:

```
400 DIM C1(10)
410 DIM D1(10)
500 MAP1 CX(10)
510     MAP2 C1,F
520     MAP2 D1,F
```

The statements at 400 and 410 produce two arrays each with ten variables. The statements at 500, 510 and 520 produce one array with twenty variables in it. The variables will still be referenced as C1(1) thru C1(10) and D1(1) thru D1(10) but their placement in memory is quite different. The C1 variables will be interlaced with the D1 variables giving C1(1), D1(1), C1(2), D1(2), C1(3),

.... C1(10), D1(10). There are also ten unformatted variables CX(1) thru CX(10) which each contain the repective pairs of C1-D1 variables in tandem. Referencing one of these CX variables will reference a 12-byte unformatted item composed of the C1-D1 pair of the same subscript. This type of formatting would be useful in sophisticated techniques only.
The following define a more complex area:

```
100 MAP1 ARRAY1
110    MAP2 UNITX(5)
120       MAP3 SIZA,B,2
130       MAP3 SIZB,B,2
140       MAP3 NTOT,F
150       MAP3 FLAG(10),B,1
160       MAP3 CNAME,S,20
170    MAP2 TOTAL,F
180 MAP1 THING,F
190 MAP1 WORK1,X,40
```

The area that is allocated by the above statements requires a total of 252 bytes of contiguous memory storage. A total of 3 levels are represented in various formats. Statement 100 defines a level 1 unformatted area called ARRAY1 which is subdivided into two level 2 items. Statement 110 defines the first of these which is an area called UNITX. The optional dimension indicates that 5 of these identical areas exist which must be referenced in the program by the subscripted variable names UNITX(1) thru UNITX(5). Each one of these areas is then further subdivided into five level 3 items (statements 120-160). Since the level 2 is subscripted because it occurs 5 times, so must each of the level 3 items be subscripted. There are 5 variables named SIZA(1) thru SIZA(5) occuring once in each of the respective variables UNITX(1) thru UNITX(5). The same holds true for the variables SIZB, NTOT, and CNAME. Statement 150, however, creates a special case since it contains a dimension also. Normally this would create an area of 10 sequential bytes referenced as FLAG(1) thru FLAG(10). In our example, however, this 10-byte area will occur once in each of the higher level areas of UNITX(1) thru UNITX(5). This implicitly then defines a double-subscripted variable ranging from FLAG(1,1) thru FLAG(5,10). Statement 170 causes the allocation to return to level 2 where one floating point variable is allocated.

The total storage requirement for the level 1 variable ARRAY1 comes out to 206 bytes as follows: 40 bytes for each of the five areas UNITX(1) thru UNITX(5) plus 6 bytes for the one variable TOTAL. Notice that since TOTAL starts a new level 2 it does not occur 5 times as do the level 3 items which comprise UNITX(1) thru UNITX(5).

Following the above mess in memory come two more variables defined in statements 180 and 190. THING is a normal floating point variable which occupies 6 bytes and WORK1 is an unformatted area whose size is 40 bytes. Note that since WORK1 was not subdivided into one or more level 2 items a size clause was required to explicitly define its storage requirements. Total storage used by the above series of statements (100-190) is 252 bytes.

Note that the variable UNITX(1) refers to the 40-byte item comprised of the variables (in order) SIZA(1), SIZB(1), NTOT(1), FLAG(1,1) thru FLAG(1,10), and CNAME(1). Moving the variable UNITX(1) to another area such as WORK1 will

transfer the entire 40-bytes with no conversions of any data. It can be seen that although this mapping system includes advanced programming practices, it provides the user with a degree of flexibility never before offered in a BASIC language implementation.

USING THE MAP STATEMENTS

Map statments may be used in immediate mode as a learning tool to see how the variables are allocated. They are not designed to be practical in the immediate mode, however, and are best used by putting them into a program file and compiling the program. In the immediate mode, if an error occurs in the syntax of the statement, the variable will have already been added to the tree and you will not be able to repeat the map statement again.

MAP statements should normally come at the beginning of the program before any references to the variables being mapped. If a reference is made to the variable before it is mapped (such as LET A = 5.8) the variable will be assigned by the normal variable allocation routines and the MAP statement will then give an error since the variable is already defined. As a convenience, all MAP1 statements force allocation to the next even byte boundary so that binary word data can be assigned properly if desired.

Due to the complexity of the syntax checking used for the MAP statement, no syntax analysis is performed on them until the program is compiled (unless the MAP statement is used in the immediate mode). If you wish to check the syntax of your progress during the entering of a group of MAP statements you may force compilation and syntax checking of your partial program with the COMPILE command. This will indicate your progress so far and also define all mapped variables up to this point so that you may interrogate them with the "@" command described in the following section.

LOCATING VARIABLES DURING DEBUGGING

Since the mapping scheme is new and fairly complex to understand fully, a command has been implemented which will assist you in locating the mapped variables and in understanding the allocation techniques used by the AlphaBasic memory mapping system. It is valid only as a system command and has no meaning if used within a program text. The command has the general format of an atsign (@) followed by a variable name. If the variable name is not followed by a subscript, the system will search for the requested variable and print out all parameters about the variable for you on the terminal. This may actually be two definitions since the variable "A" may actually be two different variables, one which is a single floating point number and one which is a subscripted array. The information returned about the variable will be the type of variable (string, binary, etc), the dimensions of the array if the variable is indeed an array, the size of the variable in bytes, and the offset to the variable from the base of the memory area used to allocate all variables. If you enter a reserved word (such as @PRINT) the system will tell you that the name is a reserved word. Remember that the current system will not allow you to begin a variable name with a reserved word so that PRINTSUM is also considered a reserved word at this time.

The general format of the definition line which is returned by the system is:

{memory-type} {variable-type} {dimensions}, {SIZE n}, {location}

Memory-type is the method of allocation used for the variable being defined. FIXED variables are those which have not been defined by a MAP statement and were allocated automatically by the compiler when they were referenced in the program. This is the normal method used by other BASIC versions to allocate variables. DYNAMIC variable arrays are those arrays which were allocated by a DIM statement or by a default reference to a subscripted variable. MAP1 through MAP16 variables and variable arrays are those which were defined in a MAP statement.

Variable-type is the type of the variable and may be UNFORMATTED, STRING, FLOATING POINT, or BINARY.

If the variable is an array the dimensions will be listed after the variable type code in the format ARRAY (n,n,n) where n,n,n are the values of the subscripts in use by the array. If the array is dynamic and has not been allocated yet the subscript values will be replaced by the letter "X" to indicate that they are not known at this point. Remember that any variable defined in a MAP statement which is in a lower level to another variable will inherit all subscripts from that higher level variable.

The size of the variable will be given in bytes. In the case of arrays, the size will represent the size of each single element within the array.

The location of the variable is a little tricky to explain since it is actually an offset to the base of a storage area set aside for the allocation of user variables. As each new variable or array is allocated it will be assigned a location which is relative to the base of this storage area. The location information is given here to help you understand the relative placement of the variables in the mapping system and does not represent the actual memory locations which they occupy. There are two distinct areas in use for variables and thus the offsets of the variables will be to one of these two areas. All FIXED and MAP1 through MAP16 variables are allocated in the fixed storage area while all DYNAMIC arrays are allocated in the dynamic array storage area. As dynamic arrays are dimensioned and redimensioned their position may shift around relative to one another and relative to the dynamic storage area base. Variables in the fixed storage area will never change position relative to each other or to the storage area base.

Array location information is given only pertinent to the base of the array itself which is the location of the first element within the array. The actual range of locations used by the array may or may not be contiguous in memory depending on whether overlapped dimensioning techniques are being used in the MAP statements. Simple (non-array) variables are defined as a location range which tells exactly where the entire variable lies within the storage area. If you want to find out where a particular element of an array is located you may follow the variable-name by the particular subscript values (decimal numbers only) of the element you wish to locate. given the two commands:

```
@A
@A(4,12)
```

The first command will give information about the array "A" while the second command will define the exact location of element A(4,12) within the array "A".

Keep in mind that this "@" command is to assist you in following the allocation of variables, particularly in more complex mapping schemes. A few minutes at the terminal with immediate mode MAP statements followed by "@" commands will help you see how the mapping scheme works.

INTERACTIVE COMMAND SUMMARY

Whenever AlphaBasic is not either compiling or executing a program it will be in interactive command mode which means it is waiting for a command from the user terminal to initiate some action. The action taken depends on the type of input entered by the user which will fall into one of the following main categories:

1. Edit command which begins with a valid source line number
2. Program statement for immediate execution (no line number)
3. Interactive system command resulting in a controlled action

Edit commands will allow the creation and editing of a source program in memory on a single line basis. The line number must be first followed by the program statement which will add the line to the source format in proper numerical sequence. If the line contains only the line number, the line will be deleted from the program. No fancy character editing commands are implemented in this version.

Program statements without line numbers result in the immediate compilation and execution of the statement entered. These program statements will be covered in another section. The remainder of this section will briefly list the available interactive commands and the corresponding action performed.

NEW

This command clears out all current source code, object code, user symbols and variables. It effectively initializes the compiler to accept new source program statements or immmediate mode statements.

LIST

The source program (if any) is listed in numerical sequence on the user terminal. If no line numbers follow the LIST command the entire program will be listed. The listing may be aborted by entering control-c on the terminal which will return the user to interactive command mode. If one line number follows the LIST command only that line number will be listed. If the command is followed by two line numbers separated by a comma, space or other non-numeric character, only those lines which fall within the range bounded by the two numbers will be listed.

DELETE

The DELETE command is used to delete groups of source lines from the program text. If the command is followed by a single line number, only that line will be deleted. If the command is followed by two line numbers separated by a comma, space or other non-numeric character, all lines of text which fall between the two line numbers inclusive will be deleted from the text.

SAVE

The entire source program is saved on the disk in the user's file area. The user must enter the name of the program (1-6 characters) following the SAVE command. The program will be saved with the extension "BAS" and will be in ASCII format which may be listed or edited with the normal system programs outside of AlphaBasic. If a previous version of the program (same name) already exists on the disk in the current user's file area that program will first be deleted before the new program is saved. No backup file will be automatically created. In actual practice, the program name may be a full system file specification if desired.

The SAVE command may also be used to save the compiled object program on disk for later running without recompilation. To save the object program the user enters the program name followed by the explicit extension "RUN" which causes the program to be compiled if necessary and then the object program to be saved on disk. The following two examples will show how the SAVE command is used to save first the source program and then the object program:

```
SAVE PAYROL                (saves source as PAYROL.BAS)
SAVE PAYROL.RUN            (saves object as PAYROL.RUN)
```

LOAD

The specified program whose name must follow the LOAD command is located on disk in the current user's file area and then loaded into memory for editing or execution. This is the reverse of the save function above. The program is expected to be in ASCII format with the default extension "BAS" unless explicitly entered as otherwise. If the program cannot be located an error message will result. In actual practice, the program name may be a full file specifier if desired.

The load command does not clear the text buffer before it loads the requested file and therefore may be used to concatenate or merge several programs or subroutines together to be saved as a single program. The separate routines must not duplicate line numbers in the other routines that they are to be merged with or else the new line numbers will overlay the old ones just as if the file had been edited in from the user terminal. The NEW command should be used prior to any load command if it is desired to insure that the text buffer is clear.

COMPILE

The current source program in memory is compiled and the object code built up in another area of memory. Control then returns to interactive command mode. The compiled program is not executed. Compilation effectively sets all variables to zero and deletes all variables that may have been generated as a direct result of immediate mode commands.

RUN

This is the normal way to initiate the running of the existing program in memory. A check is first made to see if the program has been compiled since the last editing change to the source code and if it has not been, an automatic compilation phase takes place to insure the object code is up to date. All variables are reset to zero (strings are reset to null) and the compiled object code is then executed. Execution may be interrupted at any time by typing a control-c on the user terminal. This control-c status is tested only at the beginning of each new source line so multiple statement lines will not be interrupted until all statements on that line have been completed.

CONT

The execution of the program is continued from wherever it last left off. This is normally done after a control-c interrupt, a program STOP statement, a breakpoint interrupt, or a single-step sequence. A program may not be continued after it has come to the end of the statements.

CONTROL-C

Depressing the control and c keys simultaneously will interrupt any program that is currently running and return to command interpretive mode. If a program was being executed the line number about to be executed will be printed vai the message "INTERRUPTED AT LINE nnnn". The program may be continued by the CONT or single-step commands or it may be restarted from the beginning by the RUN command.

SINGLE-STEP (line-feed key)

The single-step function is a feature not found in other versions of BASIC but is very useful in debugging programs and in teaching the principles of BASIC programming to newcomers. The single-step function causes the current line in the program to be listed on the user terminal and then executed. Any output generated by the execution of a PRINT statement will then follow on the next line. After the line has been executed the execution pointer is advanced to the next line and control returns to the user in the interactive command mode. Successive single-step commands may be used to follow the program through its paces. Single-step is legal after program STOP statements, breakpoint interrupts, control-c interrupts, and other single-step functions. Note that the single-step function is performed by hitting the line-feed key and not by actually entering the words "single-step".

BREAK

This is a feature not normally found in other versions of BASIC which allows the user to set breakpoints on one or more line numbers in a program. During execution if a line that has a breakpoint set on it is encountered the program will suspend execution and the message "BREAK AT LINE nnnn" will be printed. The system will then be in interactive command mode to allow the inspection or

changing of variable values. This suspension of execution occurs before the line that has the breakpoint set on it is executed. There is no limit to the number of breakpoints that may be set in one program. There is no additional overhead paid in execution speed when breakpoints are set. Breakpoints may be cleared by placing a minus sign in front of the line number or by compiling the program which always clears all breakpoints. If no line numbers follow the BREAK command all current breakpoints will be listed on the user terminal. For example:

BREAK	Lists all currently set breakpoints
BREAK 120	Sets a breakpoint at line 120
BREAK -120	Clears the breakpoint at line 120
BREAK 120,130,40,500	Sets breakpoints at lines 120,130,40, and 500
BREAK -50,60	Clears the breakpoint at 50 and sets one at 60

Once a breakpoint has been reached the user may optionally continue the execution of the program by either a CONT command or a single-step command. The breakpoint remains set after it has been reached until explicitly cleared by another BREAK or COMPILE command.

DELETE

The DELETE command is used to delete groups of source lines from the program text. If the command is followed by a single line number, only that line will be deleted. If the command is followed by two line numbers separated by a comma, space or other non-numeric character, all lines of text which fall between the two line numbers inclusive will be deleted from the text.

BYE

This says goodbye to basic and returns the user terminal to monitor command mode. Any program left in memory is lost forever so you may want to save it first using the SAVE command.

The source program contains statements which are executed in sequence, one at a time as they are encountered. Each of these statements normally starts with a verb followed by optional variables or statements modifiers. This section will list the program statements that are supported by AlphaBasic and give some examples where necessary for clarity.

LET

Assigns a calculated value to a specific variable during execution of the program. It is unique in that the actual word "LET" may be omitted if desired.

```
LET A5 = 12.4
LET SUM(4,5) = A1+SQR(B1)
LET C$ = "JANUARY"
A5 = 12.4
SUM(4,5) = A1+SQR(B1)
C$ = "JANUARY"
```

GOTO

The GOTO statement transfers execution of the program to a new statement location. This statement location must be either a line number or a label defined somewhere in the program. The line number or label must follow the GOTO statement in the program. The GOTO statement may be broken apart as GO TO if desired.

GOSUB - CALL - RETURN

Calls a subroutine which starts with the line number or label following the GOSUB or CALL verb. The subroutine exits via the RETURN statement which returns control to the statement following the GOSUB or CALL statement. Executing a RETURN statement without first executing a GOSUB statement will result in an error message. Both GOSUB and RETURN are currently illegal in immediate mode. Note that the CALL verb is merely another way of specifiying GOSUB for those programmers used to this verb from other languages.

ON ... GOTO

The ON GOTO statement allows multi-path GOTO branching to one of several points within the program based on the result of evaluating an expression. The actual format is:

```
ON expression GOTO point1, point2, point3, ... pointN
```

The expression can be any valid expression which will be evaluated down to a positive integer result. The result will then be tested to branch to point1 if 1, point2 if 2, point3 if 3, etc. If the result is zero, negative or greater

than pointN the program will fall through to the next statement. The points (point1 through pointN) may be line numbers, labels or any misture of the two.

ON ... GOSUB

The ON GOSUB statement parallels the ON GOTO statement in format and operation except that point1 through pointN represent entries to subroutines that will be executed based on the result of the expression evaluation. As with the GOSUB statement, the verb CALL may be used in place of the verb GOSUB giving an ON CALL statement.

READ - DATA - RESTORE

These calls allow data to be an integral part of the source program with a method for getting this data into specific variables in an orderly fashion. DATA statements are followed by one or more literal values separated by commas. String literals need not be enclosed in quotes unless the literal data contains a comma. All data statements are placed into a dedicated area in memory no matter where they appear in the source program. READ statements are followed by one or more variables separated by commas. Each time a read statement is executed the next item of data is retrived from the DATA statement pool and loaded into the variable named in the read statement. If there is no more data left in the data pool an error message results and the program is aborted. The RESTORE statement is used to initialize the reading of the data pool from the beginning again.

```
DATA 1,2,3,4,5
DATA 2.3,0.555,ONE STRING,"4,4"
READ A,B,C
READ A$
READ C(2,3),B$(4)
RESTORE
```

The READ statement is also used for reading data from random access files. For details on this refer to the section describing the file I/O system.

INPUT

Allows data to be entered from the user terminal and loaded into specific variables at execution time. The INPUT statement contains one or more variables separated by commas. When the INPUT statement is executed a single question mark is printed on the user terminal to signal the request for data entry. Numeric variables require the data to be in one of the acceptable floating point formats. String variables require the data to be an ASCII string of characters. If multiple variables are used in one INPUT statement the user is expected to enter multiple values separated by commas sufficient to satisfy the number of variables called out. If insufficient data is entered a double question mark will be printed to signal the need for additional data.

```
INPUT A1
INPUT B$,C$,Q(8)
```

If the user enters a blank line (carriage-return only) in response to a request for input, the previous values of all variables will remain unchanged and the program will proceed to the next statement. This effectively bypasses the entire input statement (or the remainder of a partial data request). If a control-c is entered in response to a request for data the input statement is bypassed (as with a carriage-return) and the program is interrupted at the next statement following the input statement. The program may be resumed by the CONT or single-step commands.

The user may cause his own prompt character or character string to be printed in place of the standard question mark by enclosing the string in quotes immediately following the INPUT verb.

```
INPUT "ENTER YOUR NAME: ",A$
INPUT "ENTER 3 VALUES FOR X, Y AND Z: ",X,Y,Z
```

The INPUT statement is also used for reading data from sequential files. For details on this refer to the section describing the file I/O system.

INPUT LINE

The INPUT LINE statement operation is identical to that of the INPUT statement with the exception that input into a string variable will accept the entire line up to but not including the carriage-return and line-feed. This allows commas, quotes, blank lines and other special characters to be input without the need for quotes around them. The INPUT LINE statement may be used in sequential file processing as well as the standard terminal input statement. The question mark prompt character is never printed for an INPUT LINE statement but the user may include his own prompt string as in the INPUT statement above. Some examples of the statement are:

```
INPUT LINE A$
INPUT LINE "ENTER YOUR FULL NAME, PLEASE: ",NAME
INPUT LINE #2, LINE'OF'INPUT
```

PRINT

The print statement performs the same as other versions of BASIC and will not be detailed extensively in this first printing. Multiple variables or literals are printed on the same line separated by commas or semicolons. Commas cause the next variable to be printed in the next zone while semicolons cause the variables to be printed with no separating spaces. If the line ends with a semicolon the carriage return will be suppressed so that the next PRINT statement executed at some later time will resume printing on the same line. For compatibility with other popular BASIC implementations, AlphaBasic also recognizes the single question mark as an alternate form of the PRINT verb.

PRINT USING is supported for formatted output and is described in another section reserved for that purpose alone.

The PRINT statement is also used for writing data to sequential files. For

details on this refer to the section describing the file I/O system.

FOR - NEXT - STEP

These statements allow the execution of loops within the program and follow the same format and restrictions as other forms of BASIC. The variable used may be subscripted if desired. If no STEP modifier is used the step value is assumed to be a positive 1. The variable name may be omitted in the NEXT statement if desired in which case the previous FOR statement will be the one that is incremented. All normal rules for nested loops and entering or exiting from within the loops apply here as in other BASIC versions. FOR and NEXT statements are illegal in immediate mode.

```
FOR A = 1 TO 10
FOR B = A1/B1 TO C1 STEP 2
FOR A = 10 TO 1 STEP -1
```

IF - THEN - ELSE

The conditional processing features in AlphaBasic give a wide variety of formats which duplicate just about all functions performed by other versions of BASIC. The formats that are acceptable are:

```
IF <relative expression> THEN <line number>
IF <relative expression> THEN <line number> ELSE <line number>
IF <relative expression> <statement>
IF <relative expression> <statement> ELSE <statement>
IF <relative expression> THEN <statement>
IF <relative expression> THEN <statement> ELSE <statement>
```

The above formats may be nested to any depth and rather than go into detail we suggest that you play around with them to determine the actual restrictions that exist. Some examples:

```
IF A=5 THEN 110
IF A>14 THEN 110 ELSE 220
IF B$="END" PRINT "END OF TEST"
IF TOTAL > 14.5 GOTO 335
IF AA=5 AND BB=6 IF CC=7 PRINT 567 ELSE PRINT 56 ELSE PRINT "NONE"
IF A=1 PRINT 1 ELSE IF B=2 THEN 335 ELSE 345
```

DIM

The dimension statement defines an array which will be allocated dynamically at execution time. There is no limit to the number of subscripts that may be used to define the individual levels within the array. The statement DIM A(20) defines an array with 20 elements referenced as A(1) through A(20). Multiple arrays may be dimensioned by a single DIM statement by separating them with commas.

Subscripts are evaluated at execution time and not at compile time thereby

allowing variables to be used as subscripts instead of fixed values. The statement DIM A(B,C) will allocate an array whose size will depend on the actual vaules of B and C at the time the DIM statement is executed.

String arrays may also be allocated such as DIM A$(5). The size of the array will depend on the current default string size in effect as specified by the last STRSIZ since each element in the array must be this number of bytes.

Arrays may be redimensioned during the execution of the program if desired. The number of subscripts must remain the same but the number of elements in each level of the array may be changed. Each redimensioning of an existing array effectively erases the old array first and then allocates a new array with all elements zeroed out.

```
DIM A(10)
DIM C(8,8), C$(10,4)
DIM TEST(A,B*4)
DIM A(B(4))
```

SIGNIFICANCE

The significance statement allows the user to dynamically change the default value of the numerical significance of the system for unformatted printing. The significance value can be any value from 1 through 11 and will represent the maximum number of digits to be printed in unformatted numbers. Rounding off to the specific number of digits will be performed only prior to the actual printing of the result. The statement SIGNIFICANCE 8 will set the number of printable digits to 8. The value is interpreted at run time and therefore may be any valid numeric expression including variables if desired. The current significance of the system is ignored when PRINT USING is in effect.

Note that the signifcance statement only affects the final printed result of all numeric calculations. The calculations themselves and the storage of intermediate results is always performed in full 11-digit precision to minimize propagation of errors.

The significance of the system is initially set at 6 digits when the system is first started. This is equivalent to standard single-precision formats used in most of the popular versions of BASIC. The significance is not reset by the RUN command and therefore may be set in immediate mode just prior to the actual running of a test program. Of course, any SIGNIFICANCE statements encountered during the execution of the program will reset the value.

STRSIZ

The string size statement sets the default value for all strings which are encountered for the first time during the compilation phase. Initially, the default value of all strings in the absence of a STRSIZ statement is 10 bytes. The statement STRSIZ 25 would cause all newly allocated strings which follow to have a maximum size of 25 bytes instead of 10 bytes. This includes the allocation of string arrays. The size value is evaluated at compilation time and therefore must be a single positive integer.

RANDOMIZE

Resets the random number generator seed to begin a new random number sequence starting with the next RND(x) function call.

STOP

Causes the program to suspend execution and print the message "PROGRAM STOP AT LINE nnnn" and then return control to the user in the interactive command mode. The user may then continue with the next statement in sequence by executing a CONT command or with single-step commands.

END

Causes the program to terminate execution and return to the READY mode. The END statement does not terminate compilation of the program nor is it required at the end of the program.

OTHER VERBS

AlphaBasic supports other verbs which are described in separate sections to more fully go into the details of their operation. The following verbs are described elsewhere:

SCALE - scaled arithmetic modifier
CHAIN - executes a new program or command file
ON ERROR - controls error trapping and processing
OPEN - opens an I/O file for processing
CLOSE - closes an I/O file to further processing
WRITE - write a record to a random access file
KILL - deletes a file from disk
ALLOCATE - allocates a random access file on disk
LOOKUP - searches for a file and returns its size
XCALL - executes an external assembly language subroutine
ISAM - facilitates processing of indexed sequential files

The following is a list of the currently implemented functions which are available for use in expressions. Note that the mode independence feature of the expression processor will perform automatic conversions if a numeric argument is used where a string argument is expected and vice versa.

EXP(X)

Returns the constant e (2.71828) raised to the power X.

LOG(X)

Returns the natural (base e) logarithm of the argument X.

LOG10

Returns the decimal (base 10) logarithm of the argument X.

SQR(X)

Returns the square root of the argument X.

INT(X)

Returns the largest integer less than or equal to the argument X.

FIX(X)

Returns the integer part of X (fractional part truncated).

FACT(X)

Returns the factorial of X.

ABS(X)

Returns the absolute value of the argument X.

SGN(X)

Returns a value of -1, 0 or 1 depending on the sign of the argument X. Gives -1 if X is negative, 0 if X is 0 and 1 if X is positive.

RND(X)

Returns a random number generated by a pseudo-random number generator based on a previous value known as the "seed". The argument X controls the number to be returned. If X is negative it will be used as the seed to start a new sequence of numbers. If X is zero or positive the next number in the sequence will be returned depending on the current value of the seed (this is the normal mode). The RANDOMIZE statement may be used to create a seed which is truly random and not based on a fixed beginning value set by the system.

MEM(X)

Returns a positive integer value which specifies the number of bytes currently in use for various memory areas used by the compiler system. The most common use of this is with an argument of 0 which returns the number of free bytes left in the user memory partition. This MEM(0) call duplicates the action performed by the FRE(X) function in other versions of BASIC. Other values of the argument X return memory allocations which pertain to various areas in use by the compiler and may or may not be of use to you. The byte counts returned for the various values of X are:

0 - Free memory space remaining in current user partition
1 - Total size of current user partition
2 - Size of source code text area
3 - Size of user label tree
4 - Size of user symbol tree (variable names and user function names)
5 - Size of compiled object code area
6 - Size of data pool resulting from all compiled DATA statements
7 - Size of array index area (dynamic links to variable arrays)
8 - Size of variable storage area (excluding arrays)
9 - Size of file I/O linkage and buffer area

Note that the parameters for values above 1 may change as new versions of the compiler are developed. Also, some of these values will be meaningless when running the runtime object module in compiled mode.

EOF(X)

The EOF function returns a value giving the status of a file whose file number is X. The file is assumed to be open for sequential input processing. The values returned by the EOF function are:

-1 if the file is not open or the file number X is zero
0 if the file is not yet at end-of-file during input calls
1 if the file has reached the end-of-file condition

Due to the method used by the AMOS operating system for processing files, the EOF status will not be valid until after an INPUT statement which reaches the end-of-file condition. Any INPUT statements which reach EOF will return numeric zero or null string values forever more. This means that the normal sequence

for processing sequential input files would be to INPUT the data into the variables and then test the EOF(X) status before actually using the data in those variables.

EOF should only be tested for sequential input files. Files open for output or for random processing will always return a zero value.

LEFT(A$,X) or LEFT$(A$,X)

Returns the leftmost X characters of the string expression A$.

RIGHT(A$,X) or RIGHT$(A$,X)

Returns the rightmost X characters of the string expression A$.

MID(A$,X,Y) or MID$(A$,X,Y)

Returns the substring composed of the characters of the string expression A$ starting at the Xth character and extending for Y characters. A null string will be returned if X > LEN(A$).

LEN(A$)

Returns the length of the string expression A$ in characters.

INSTR(X,A$,B$)

Performs a search for the substring B$ within the string A$ beginning at the Xth character position. It returns a value of zero if B$ is not in A$ or the character position if B$ is found within A$. Character position is measured from the start of the string with the first character position represented as one.

ASC(A$)

Returns the ASCII decimal value of the first character in string A$.

CHR(X) or CHR$(X)

Returns a single character string having the ASCII decimal value of X. Only one character is generated for each CHR or CHR$ function call.

STR(X) or STR$(X)

Returns a string which is the character representation of the numeric expression X. No leading space is returned for positive numbers.

VAL(A$)

Returns the numeric value of the string expression A$ converted under normal BASIC format rules.

SPACE(X) or SPACE$(X)

Returns a string of X spaces in length.

TRIG FUNCTIONS

The following trig functions are implemented in full 11-digit accuracy:

SIN(X)	Sine of X
COS(X)	Cosine of X
TAN(X)	Tangent of X
ATN(X)	Arctangent of X
ASN(X)	Arcsine of X
ACS(X)	Arccosine of X
DATN(X,Y)	Double arctangent of X,Y

FORMATTED OUTPUT VIA PRINT USING STATEMENTS

The PRINT USING statement is an extension of the standard PRINT statement which allows the output to be formatted into specific character positions suitable for business reports, formal text applications, and the like. The format of the statement appears as follows:

```
PRINT USING <string>, <list>                (output to terminal)
PRINT #<file>, USING <string>, <list>       (output to file)
```

The string expression is used to control the formatting of the variables as they are enrountered in the print list and must match the format of the variables to be printed. The string may be either a string constant, a string variable, or a string expression which is interpreted as an exact image of the line to be printed. The list is the sequence of variables or expressions to be printed using all the rules of the standard PRINT statement. All characters in the formatting string will be printed as they appear except for the special formatting characters which will be described below. The string is continually scanned over and over until the list of print items is exausted. The formatting characters and their usage in the string are described in the following paragraphs:

EXCLAMATION MARK

An exclamation mark (!) in the format string causes the first character of the corresponding string variable to be printed in the corresponding space. The rest of the string variable if it exceeds one character will be ignored.

BACKSLASHES

Two backslashes (\) in the format string define a string field whose size equals the number of characters enclosed by the brackets plus the brackets themselves. Normal BASIC syntax dictates that the characters between the backslashes be spaces but AlphaBasic will accept any characters. If the string variable to be printed exceeds the size of the specified string field the excess characters will be ignored. If the string variable is shorter than the specified string field, trailing blanks will be added to fill out the correct field size.

NUMERIC FIELDS

Digit positions within formatted numeric fields are specified in the format string by the pound-sign (#) using one for each position desired, both in front of and behind the decimal point. One decimal point is allowed to define the explicit alignment of the digits within the numeric field format. Normally, numeric fields are right justified with leading blanks being used to fill in for digit positions that are not required in front of the decimal point. Unused digit positions behind the decimal point are filled with trailing zeros. If the numeric field specified in the format string is too small to contain the numeric variable to be printed, the field will be printed with a leading percent-sign (%) indicating the overflow. This will be followed by the number in standard BASIC format.

If the format field specifies any digit positions in front of the decimal point, at least one digit is always output before the decimal point itself. If necessary, this digit is a zero.

Note that other special characters (described in the following paragraphs) also define numeric digit positions in addition to performing special formatting functions.

ASTERISK FILL

If a numeric field in the format string begins with a double asterisk (**) any leading spaces that would normally be output in front of a number will be replaced by asterisks. This is quite useful in printing checks, for instance. The double asterisk also defines two digit positions.

FLOATING DOLLAR SIGN

If a numeric field in the format string begins with a double dollar sign ($$) a dollar sign will be printed immediately preceding the first digit of the number. The double asterisk also defines two digit positions, one of which is taken up by the dollar sign itself.

The above two special functions may be combined by starting the numeric format field with the symbols "**$" combines the asterisk fill and floating dollar sign functions described above. This combination also defines three digit positions, one of which is taken up by the dollar sign itself.

TRAILING MINUS SIGN

If a numeric field is terminated by a minus sign in the format string, the sign of the output number is printed following the number instead of before it. If the number is negative, a minus sign will be printed. If the number is zero or positive, a blank will be printed. Note that if the trailing minus sign is not used, space must be provided in the numeric field for the sign to precede the number if it is negative.

COMMA

If the numeric format field contains one or more commas in front of the decimal point, a comma will be inserted every three digits to the left of the decimal point when the number is printed. Each comma also defines one digit position in the format field. Commas appearing after the decimal point will be treated as printing characters.

EXPONENTIAL FORMAT

Exponential format may be specified by following the numeric field designation with four up-arrows (^^^^) which defines the space taken up by the E NN exponent value. As with other numerical formats, any decimal point arrangement is allowed and the significant digits are left justified with the exponent being adjusted as necessary.

AlphaBasic uses a floating point format which gives an accuracy of 11 significant digits. Unfortunately, this accuracy is absolute only when dealing with those numbers which are total integers in the first 11 positions to the left of the decimal point. This fact stems from the conversions that are required from decimal input to the binary floating point format used in the hardware. For most business users, the actual range of numbers contains two digits to the right of the decimal point and nine digits to the left of the decimal point. When the fractional part of the number is converted between decimal and binary formats, a small but significant error is sometimes introduced which may propogate into large inaccuracies when dealing with absolute dollars-and-cents values.

AlphaBasic incorporates a scaling feature which helps to alleviate these problems by storing all floating point numbers with a scale offset. This offset may be used to effectively designate where the 11 absolute accuracy digits are located in relation to the decimal point. This is done by multiplying every input number by the scaling factor and then dividing it out again before printing. This is a simplified explanation and many other checks and conversions are done internally to scaled numbers but that is the general idea.

The scaling factor represents the number of decimal places that the 11-digit "window" will be effectively shifted to the right in any floating point number. For example, the most common application is in a business environment where the scaling factor of 2 would be used to give absolute 11 place accuracy which extends 2 places to the right of the decimal point. This means that the value of 50.12 will be multiplied by the scaling factor of 2 digits (100) and stored as the floating point value of 5012. Since this value is an integer, it has absolute accuracy. Just before printing this number it will be divided by the scaling factor to reduce it to its intended value of 50.12 and everybody is happy.

Other little conversions had to be included into the system to take care of all the little subtle effects of storing scaled numbers. For example, when converting scaled numbers to integer or binary format, the number must be unscaled first before conversion. When two scaled numbers are multiplied together the result is a number which must be unscaled once. Division of two scaled numbers creates exactly the opposite problem. Dealing with scaled numbers for exponential, logarithmic and trigonometric functions creates even more exotic problems. All these conversions are done automatically by AlphaBasic and so the programmer is relieved of the task of keeping track of them.

Scaled arithmetic will normally be entered at the start of a program and will continue in effect throughout the program. The statement for setting the program into scaled mode is:

SCALE n

The scaling factor "n" must be a decimal digit in the range of -30 to +30 and may not be a variable since scaling is done at compile time for constant values as well as at runtime for input and output conversions. Negative scaling moves

the 11-digit window to the left and for most cases will not be of use to the average programmer.

A few words of caution are in order here. Once the SCALE statement has been detected during compilation, all constant values that follow are scaled by the scaling factor so that they are stored properly. In addition, a runtime command is generated in the executable program which causes the actual scaling to be performed on input and print values when the program is running. If two or more different SCALE statements are executed in the same program, some very strange results may come out unless the user is totally familiar with what is happening in regards to compile time and run time conversions. We suggest that you play with this one a bit before delving into it full steam.

One other word of caution. Floating point numbers that are stored in files by the sequential output PRINT statement will be unscaled and output in ASCII with no problems. Floating point numbers that are written to random access files by using the WRITE statement will not be unscaled first and any program that reads this file as input had better be operating in the same scaling mode or else apply the scale factor explicitly to all values from the file. Binary and string values, of course, are never modified regardless of the scaling factor currently in use.

ALPHABASIC FILE I/O SYSTEM

AlphaBasic supports both sequential and random access disk files. Data may optionally be written in ASCII or packed binary formats. Files created by AlphaBasic programs are compatible with all other system utility formats and may be interchangeably introduced into and manipulated by programs written in other languages. Conversely, files created by other languages and system utilities may be read and manipulated by programs written in AlphaBasic.

Files are created and referenced by the general statements OPEN, CLOSE, INPUT, PRINT, READ, and WRITE. All file references are done by a file number which may be any legal integer value from 1 to 65535. There is no absolute limit to the number of files that may be open at any given time in a program but since each file requires a certain amount of memory there is a practical limit to this number based on available memory. The file number always follows the verb in any file I/O statement and may be any legal numeric expression which is preceded by a pound sign (#). File number zero is defined as the user terminal and is legal in file statements to allow generalized programs to be written which may selectively output to either a file or to the terminal at run time.

All open files are automatically closed (if not closed explicitly by a CLOSE statement) when the program exits or when a CHAIN statement is executed. No two files may be opened with the same file number at the same time but after a file has been closed another file may be reopened using the same file number if desired. All file statements are valid in immediate mode but any open files are automatically closed before each new RUN command is executed thereby preventing files which were opened by an immediate statement to be written or read by statements within an executable program. Under the current version of AlphaBasic, each open file requires about 580 bytes of free memory for buffers and control blocks. Future releases will allow more than one file to share the same buffer area thereby reducing the execution memory requirements.

SEQUENTIAL ASCII FILES

Sequential disk files are the easiest to understand and to implement in AlphaBasic. Data is written in ASCII format and all numerics are stored as ASCII string values. Carriage-returns and line-feeds are included in the output file as a result of the print statement formatting but are bypassed when the file is read by another program. The data files created by AlphaBasic sequential I/O functions normally have the extension "DAT" unless otherwise explicitly stated in the OPEN statement. These files are normal ASCII sequential files in all respects and may be manipulated by the text editor, the print spooler, or any of the other system utilities.

Data is written to sequential ASCII files by using the PRINT statement with a file number (non-zero) following the verb in standard I/O statement format. Data is read back from sequential ASCII files by using the INPUT statement in a similar manner.

RANDOM ACCESS FILES

Random access files are more complex than sequential files but offer a much more flexible method for storing and retrieving data in different formats. Random files are written in what is considered unformatted or packed data mode. All program accesses to random files are made via the "logical record" approach. A logical record is defined as a fixed number of bytes whose format is explicitly under control of the program performing the access. Physical records on the disk are each 512 bytes long and each random file must be preallocated as some given number of these 512-byte physical records. Logical records may be any length from 1 byte to 512 bytes in length. The AlphaBasic I/O system will automatically compute the number of logical records that will fit into one physical record and perform the blocking and unblocking functions for you. For example, if your logical record size is defined as 100 bytes, then each physical record on the disk will contain 5 logical records with the last 12 bytes of each physical record being lost. Therefore, the most efficient use of random files comes when the logical record size will evenly divide into 512 bytes (32, 64, 128, etc).

Random access files are preallocated once using the ALLOCATE statement and giving the number of physical 512-byte records to be allocated. It is up to the programmer to calculate the maximum number of logical records required in the file and then, using the above description, calculate how many physical records will be required to completely contain the number of logical records desired. For instance, assume the logical record size is 100 and you need a maximum of 252 logical records in your file. Each physical disk record is 512 bytes and therefore will contain 5 logical records. You need 252 logical records so dividing 252 by 5 gives 50 full physical records plus 2 logical records remaining. Since the file must be allocated in whole physical records you will need 53 physical records which will give you a maximum of 255 logical records. These logical records will be referenced in your program as records 0 through 254 since the first record of any random file is numbered record 0.

The logical record size is specified dynamically in the OPEN statement when the file type is RANDOM so it is possible to get things fouled up if you do not have the record size correct. No logical record size is maintained within the file structure itself. This fact does make it nice in one respect and that is that a file which is accessed by many programs can have its record size expanded without recompiling all programs. Heres how: Assume you have a file which is considered the parameter descriptor file for all other files in the entire system. This file gives the record size as 100 bytes for the vendor name and address file (as an example). All programs which reference the vendor file first read this parameter file to get the size of the vendor file logical record. The programs then set the size into a variable and use this variable in the OPEN statement for the record size. Each READ or WRITE call will then manipulate the 100 bytes of data by reading or writing to or from variables whose size totals 100 bytes. Lets say you now want to expand the file to 120 bytes and that most of the programs will not have to make use of the extra 20 bytes until some time in the future. You write a program which copies the 100-byte file into a new 120-byte file and update the main parameter file to indicate that the new record size for the vendor file is 120 bytes instead of 100. Each program will now open the file using the new 120-byte record size (since it is read in from the parameter file at runtime) but will only READ or WRITE the first 100 bytes of each record due to the variables used by the READ and WRITE calls. Got the message?

OPEN

In order to manipulate data to or from a file it must be opened first. The open statement assigns a unique file number to a file and also specifies the name that is to be given to an output file or to be used in the locating of an input file. The general format is:

```
OPEN #<file>, <filename>, <mode>, {record-size, record-number-variable}
```

file - any numeric expression which evaluates to an integer from 0-65535 (0 is defined as the user terminal and treated as such)

filename - any string expression which evaluates to a legal file description

mode - specifies the mode for opening the file:
INPUT - opens an existing file for input operations
OUTPUT - creates a file for output operations
RANDOM - opens an existing file for random read/write

The remaining two options must be used for RANDOM mode only:

record-size - an expression which specifies dynamically at runtime the logical record size for read/write operations on the file

record-number-variable - a non-subscripted numeric variable which must contain the record number of the desired random access for READ or WRITE statements when they are executed

Any attempts to read or write to a file which has not been opened will result in an error message and the program will be aborted. The filename string may be as brief as the name of the file in which case it is assumed to have an extension of "DAT" and reside in the current user's disk file area. The filename string may expand to become a complete file specification if desired giving the explicit location of the file in another user area and even on another disk drive. Some examples are:

```
OPEN #1, "DATFIL", INPUT
OPEN #15, "PAYROL.TMP", OUTPUT
OPEN #A, C$, OUTPUT
OPEN #3, "DSK1:OFILE.ASC[200,20]", OUTPUT
OPEN #1, "VENDOR.DAT", RANDOM, 100, RECNUM
```

The OPEN statement is the only statement which references the file by its actual ASCII filename in the standard operating system format. All further references in the program are made by the file number which is assigned in the OPEN statement #<file> expression.

CLOSE

The CLOSE statement terminates the processing of data to or from a file. Once a

file has been closed, no further references are allowed to that file number until another OPEN statement is executed. Any files that are still open when the program exits will be closed automatically. The format of the CLOSE statement is:

CLOSE #<file>

KILL

The KILL statement erases one file from the disk. It does not need a file number and no open or close need be performed to KILL a file. The format for the KILL statement is:

KILL <filename>

As in the OPEN statement, the filename is any string expression which evaluates to a legal file description.

LOOKUP

The LOOKUP statement looks for a file on the disk and returns a flag which tells you if the file was found and if so, how many records it contains. The format for the statement is:

LOOKUP <filename>, <result-variable>

As in the OPEN statement, the filename is any string expression which evaluates to a legal file description. The result-variable is any legal numeric variable which will receive the result of the search. If the file was not found, a zero will be returned. If the file was found and is a sequential file, a positive number will be returned which is the number of records in the file. If the file was found and is a random (contiguous) file, a negative number will be returned which is the number of records in the file. In either case, the number of records represents physical 512-byte disk records and must be divided by the blocking factor for random files if the number of logical records is desired.

ALLOCATE

The ALLOCATE statement is used to preallocate a contiguous file on disk which may then be opened for random processing. An attempt to allocate a file which already exists will result in an error message. A random file need only be allocated once and may then be opened for random read/write operations as many times as desired. The statement format is:

ALLOCATE <filename>, <number-of-records>

As in the OPEN statement, the filename is any string expression which evaluates to a legal file description. The number-of-records is a numeric expression which represents the number of physical 512-byte disk records to be allocated to the file.

INPUT

Once a file has been opened for input, the data is read from the file by a special form of the INPUT statement using a file number which corresponds to the number assigned in the OPEN statement. The variables in the list may be either numeric or string variables but must follow the format of the data in the file being read. Weird results will occur if you attempt to read string data into a numeric variable or vice-versa. The general format of the INPUT statement is:

INPUT #<file>, <variable-1>, <variable-2>, ... <variable-n>

During the reading of the input data into the variable list all leading spaces will be bypassed unless enclosed within quotes just as in the normal form of the INPUT statement. Also, all carriage-returns and line-feeds will be bypassed allowing the file created by the PRINT statements to contain formatted line data if desired. Commas, spaces, and end-of-line characters will all terminate numeric data strings and will then be bypassed.

PRINT

Once a file has been opened for output, the data is written to it by a special form of the PRINT statement using a file number which corresponds to the number assigned in the OPEN statement. All techniques usable in the normal form of the PRINT statment which outputs to the terminal may be used in the file form including PRINT USING for formatted data. The data which is output to the file is in the exact format that would appear on the user terminal if the file number had been omitted. The general format of the PRINT statement along with some valid examples follow:

```
PRINT #<file>, <data-list>

PRINT #1, A, B, C
PRINT #4, USING A$, A, SQR(A)
PRINT #Q1, USING "###.##", A1(10);
PRINT #1, "THIS IS A SINGLE LINE"
```

READ

The READ statement is used to read a selected logical record from a file which has been opened for random access processing. The logical record which is transferred by the system I/O is that whose record number is currently in the variable mentioned in the OPEN statement. The format of the READ statement is:

READ #<file>, <variable-1>, <variable-2>, ... <variable-n>

The variables in the list may be any format but they obviously should match that of the designated record format. The data will be read into the variables as unformatted bytes without regard to variable type. The data will be transferred into each variable until the variable has been completely filled and then the next variable in the list will be filled, and so on. If the record is longer than the variable list specifies, all excess data in the record will not be

transferred. An attempt to transfer more data than is in the logical record size will result in an error message. The most efficient use of the random files comes when the variable or variables used are mapped by the MAP statement to the exact picture of the record format in use.

WRITE

The WRITE statement is used to write a selected logical record into a file which has been opened for random access processing. The logical record which is trnasferred by the system I/O is that whose record number is currently in the variable mentioned in the OPEN statement. The format of the WRITE statement is:

WRITE #<file>, <variable-1>, <variable-2>, ... <variable-n>

The variables in the list may be any format but they obviously should match that of the designated record format. The data will be written into the logical record from the user variables as unformatted bytes without regard to variable type. The data will be transferred from each variable until the variable has been completely emptied and then the next variable in the list will be used, and so on. If the record is longer than the variable list specifies, all excess data in the record will not be modified. An attempt to transfer more data than is in the logical record size will result in an error message. The most efficient use of the random files comes when the variable or variables used are mapped by the MAP statement to the exact picture of the record format in use.

CALLING EXTERNAL ASSEMBLY LANGUAGE SUBROUTINES

External subroutines written in assembly language code may be called from any AlphaBasic program using the XCALL statement. The syntax for this statement is as follows:

```
XCALL routine,argument-1,argument-2,....argument-n
```

The routine to be called is an assembly language program which has been assembled using the MACRO assembler. The resulting PRG program file must then be renamed to give it the assumed extension SBR indicating this is a subroutine and not a runnable program. When the XCALL statement is executed by the AlphaBasic runtime system, the named subroutine will be located in memory and then called as a subroutine. AlphaBasic first saves all registers and then sets certain parameters into these registers for use by the external subroutine. The addresses of the arguments are calculated and entered into an argument list in memory along with their sizes and type codes. The base address of this list is then passed to the user routine in register R3.

The arguments may be one of two basic forms. A variable name may be used in which case the argument entry in the list will reference the selected variable within the user impure area. This variable is available to the called subroutine for both inspection and modification. The argument may also be an expression (numeric or string) in which case the expression is evaluated and the result is placed on the arithmetic stack (referenced by R5). This result is then referenced in the argument list entry instead of a single variable. It is available for inspection only since the stack is cleared when the subroutine exits.

The user routine is free to use and modify all six general work registers (R0-R5) and may use the stack for work space as required. When the subroutine has completed its execution a return must be made to the runtime system by executing the RTN subroutine return instruction.

REGISTER PARAMETERS

The following registers are set up by the runtime system to be used as required by the external subroutine. They may be modified if desired as they have been saved before the subroutine was called.

R0 - indexes the user impure variable area. R0 is used throughout the runtime system to reference all user variables. Details on the format of this area are not available at this time and the user need not be concerned with them. R0 may be used as a work register.

R3 - points to the base of the argument list. R3 may be used to scan the argument list for retrieval of the argument parameters.

R4 - points to the base of the free memory area that may be used by the external subroutine as work space. This is actually the address of the first word following the argument list in memory

and may be used to store a terminator word to stop scanning of the argument list if desired.

R5 - this is the arithmetic stack index used by the runtime system. The arithmetic stack is built at the top of the user partition and grows downward as items are added to it. When the external subroutine is called, R5 points to the current stack base. Since the arithmetic stack may contain valid data, the external subroutine must not use the word indexed by R5 or any words above that address.

ARGUMENT LIST FORMAT

The list of arguments specified in the XCALL statement may range from no arguments at all to a number limited only by the space on the command line. To pass these arguments to the external subroutine, an argument list is built in memory which describes each variable named in the list and tells where it can be located in the user impure area. The variables themselves are not actually passed to the subroutine, but rather their absolute locations in memory are. In this way, the subroutine may inspect them and modify them directly in their respective locations. This does not apply to expressions which are built on the stack as described previously.

R3 points to the first word of the argument list which is a binary count of how many arguments were contained in the XCALL statement. Following this count word comes one 3-word descriptor block for each argument specified. If there are no arguments in the XCALL statement, the argument list will consist only of the single count word containing the value of zero.

The format of each 3-word block describing one argument is as follows:

Word 1 - variable type code. Bits 0-3 contain the hex type code for the specific variable: 0=unformated, 2=string, 4=floating point, 6=binary, 8 through E are currently unassigned. Bit 4 is set to indicate the variable is subscripted or clear to indicate the variable is not subscripted. Other bits in the type code word are meaningless.

Word 2 - absolute address of variable in user impure area. This address is the first byte of variable no matter what is type or size might be.

Word 3 - size of the variable in bytes.

Note that the above descriptions also apply to the expression arguments with the exception that the results will be located above the address specified by R5 instead of below it.

The argument list is built in free memory directly above the currently allocated user impure area. R4 points to the word immediately following the last word in the argument list. The user may scan the argument list and determine its end either by decrementing the count word at the base of the list or by scanning until the scan index reaches the address in R4.

FREE MEMORY USAGE

When the subroutine is called, indexes R4 and R5 mark the beginning and end of the free memory that is currently available for use as workspace. This area is not preserved by the runtime system and the subroutine must not count on its security between XCALL statements. Note that the word at @R4 may be used as the first word but the word at @R5 is the base of the arithmetic stack and must not be destroyed. The last word of free memory is actually at -2(R5) for all practical purposes.

The runtime system has its own internal memory management system and does not conform to the AMOS operating system memory management. Therefore, the external subroutine must not use the GETMEM monitor calls to generate a block of work space in memory. Also, if any file calls are to be done they must be done with internal buffers since the INIT call sets up a buffer by using the GETMEM monitor call.

SUBROUTINE LOADING

The version 3.1 release requires that the subroutines being called by the BASIC program already exist either in system memory or user memory before beginning execution of the program. These subroutines must be loaded into system memory by the SYSTEM call in the SYSTEM.INI file or they must be loaded individually into the user memory partition with the LOAD command while in monitor command mode. Note that this is the monitor LOAD command to load the SBR module which is vastly different from the LOAD command used to load a basic source file once you are running AlphaBasic.

Future releases will provide capabilities for loading subroutines and overlays from within the BASIC program itself.

ERROR TRAPPING

AlphaBasic allows the user program to trap errors that would normally cause the system to print an error message and abort the program run. During interactive processing this would return you to AlphaBasic command mode and during compiled run processing it would return you to to monitor command mode. Use of the ON ERROR GOTO and RESUME statements allows these errors to be detected within the user program and immediate action to be taken when appropriate.

ON ERROR GOTO STATEMENT

Error trapping is enabled and disabled by using the ON ERROR GOTO statement in one of two forms. The first form specifies a line number (or label) within the program. When the program encounters this ON ERROR statement it stores the line number and sets a flag enabling error trapping. If an error occurs at any time after this control will be transferred to the routine specified by the line number. No error message will be printed. Examples of this form of the statement are:

```
ON ERROR GOTO 500
ON ERROR GOTO TRAP'ROUTINE
```

The error routine must then take appropriate action based on the type of error which caused the trap. The ERR function will return the following data based on conditions at the time of the error:

```
ERR(0) = numeric code specifying the type of error detected
ERR(1) = last line number encountered prior to the error
ERR(2) = last file number accessed (pertinent only for file errors)
```

The second form of the statement disables further error trapping by specifying a line number of zero or leaving the line number off completely.

```
ON ERROR GOTO 0
ON ERROR GOTO
```

After executing the above form the program will print the standard error message and abort the program run. A special case exists when the above statement is encountered within an error recovery routine (prior to executing the RESUME statement). In this instance the error trapping is disabled and the existing error is forced to be processed as if no error trapping were ever enabled. It is recommended that all error trapping routines execute the ON ERROR GOTO 0 statement for all errors which have no special recovery processing.

Note that if an error occurs within the error trapping routine itself that error will be forced and the standard error message will occur. There is no method to detect errors within the error recovery routine.

RESUME STATEMENT

The RESUME statement is used to resume execution of the program after the error recovery procedure has been performed. The statement takes on two forms similar to the ON ERROR GOTO statement. The first form specifies a line number (or label) wihtin the program at which point the execution is to be resumed:

```
RESUME 410
RESUME TRY'AGAIN
```

The second form specifies a line number of zero or no line number at all and causes the execution to be resumed with the statement that caused the error to occur.

```
RESUME 0
RESUME
```

Both forms cause the error condition to be cleared and error trapping to be enabled again.

CONTROL-C TRAPPING

When the operator types a control-c on his keyboard during the execution of an AlphaBasic program the program is suspended at the next statement. Action taken then depends upon the status of the error trapping flag. If no error trapping is enabled the program is aborted with the appropriate message being printed on the terminal. If error trapping is enabled the error trapping routine is entered with the code in ERR(0) being set to 1. This feature allows the user to prevent inadvertant aborting of programs during critical times such as file updates.

Control-c action is suspended during error recovery processing to prevent accidental aborting of the program due to an error condition occurring within the error routine. The control-c will be detected immediately upon execution of the RESUME statement.

ERROR CODES RETURNED BY ERR(0)

Code	Meaning
1	Control-c interrupt
2	System error
3	Out of memory
4	Out of data
5	NEXT without FOR
6	RETURN without GOSUB
7	RESUME without ERROR
8	Subscript out of range
9	Floating point overflow
10	Divide by zero
11	Illegal function value
12	XCALL subroutine not found
13	File already open
14	IO to unopened file
15	Record size overflow
16	File specification error
17	File not found
18	Device not ready
19	Device full
20	Device error
21	Device in use
22	Illegal user code
23	Protection violation
24	Write protected
25	File type mismatch
26	Device does not exist
27	Bitmap kaput
28	Disk not mounted
29	File already exists
30	Redimensioned array
31	Illegal record number

SYSTEM FUNCTIONS

AlphaBasic supports a unique group of operators called system functions which provide the programmer with the ability to get to the I/O ports, physical memory (sometimes referred to as PEEK and POKE) and various system parameters. The syntax of a system function parallels that of a standard function with the reserved word representing the desired function followed by optional arguments enclosed within parenthesis. The major difference is that a system function may appear on the left side of an assignment statement whereby it represents an output or write condition to the system function. System functions used within expressions on the right side of an assignment statement will perform an input or read operation and deliver back a result to be used in the expression evaluation.

IO(X)

The IO system function allows the 256 I/O ports to be selectively read from or written to. In both cases only one byte will be considered and an output expression greater than 255 will merely ignore the unused bits.

```
IO(X) = <expr>      !writes the low byte of expr to port X
A     = IO(X)       !reads port X and places the result into A
```

BYTE(X) and WORD(X)

The BYTE and WORD system functions allow the programmer to inspect and alter any memory locations within the 65K memory addressing range of the machine. These operations have often been called PEEK and POKE statements in other implementations of BASIC. The BYTE functions will deal with 8 bits of data in the range of 0-255 and the WORD functions will deal with 16 bits of data in the range of 0-65535 inclusive. Any unused bits will be ignored with no error message. Note that these commands are not protected and it is possible to poke the operating system, other users or yourself to death with improper use.

```
BYTE(X) = <expr>    !writes the low byte of expr into memory loc X
WORD(X) = <expr>    !writes the low word of expr into memory loc X
A       = BYTE(X)   !reads memory loc X and places the byte into A
A       = WORD(X)   !reads memory loc X and places the word into A
```

TIME

The TIME system function requires no argument and is used to set and retrieve the time of day as stored in the system monitor communications area. The time is stored as a two-word integer representing the number of clock ticks since midnight. The programmer is responsible for conversions to printable format in those cases where it is required. One clock tick represents one interrupt from the CPU line clock which is usually 60 hz for domestic systems and 50 hz for overseas systems. Dividing the time by the clock rate will give the number of seconds since midnight. Converting this to current time is then accomplished by successive division by 60 to get minutes and again by 60 to get hours.

```
TIME = <expr>          !sets time-of-day in system to expr
A    = TIME            !returns time-of-day in clock ticks into A
```

DATE

The DATE system function is identical to the TIME function except that it sets and returns the two-word system date. There is no current format defined by Alpha Microsystems for this date and it may be stored in any format you choose. Some common methods are to pick a base date (say 1/1/60) and store the date as the number of days since that date. Another method is to store the Julian date with the year being offset by the appropiate integer amount. The date will store a positive value of 2^32 or greater than 4 billion.

```
DATE = <expr>          !sets system date to expr
A    = DATE            !returns system date into A
```

```
1  !
2  ! Demo of use of Alpha Basic DATE routine with "DATE" formatted date.
3  !
10 MAP1 DATE'HOLDER,B,3
11 MAP1 MMDDYY,,,,@DATE'HOLDER
12  MAP2 MM,B,1
13  MAP2 DD,B,1
14  MAP2 YY,B,1
15 PRINT
16 !
20 DATE'HOLDER = DATE
21 PRINT USING "Current date is ##/##/##", MM,DD,YY
22 PRINT
30 !
32 INPUT "Enter new date MM,DD,YY : ", MM, DD, YY
34 DATE = DATE'HOLDER
36 PRINT
38 GOTO 20
```

EXPANDED TAB FUNCTIONS

The TAB function in AlphaBasic has been expanded past the normal usage to include terminal screen handling such as cursor control and other special functions. To be used only in a PRINT statement, the TAB function operates in the traditional manner when supplied with only a single numeric argument such as TAB(X). In this case the function causes the carriage to be positioned over to the "X" column on the current line. When supplied with two arguments such as TAB(X,Y), however, the TAB function performs special CRT functions.

If the value of X is positive the X,Y arguments will be treated as (row,column) coordinates for positioning the cursor on the terminal screen. The following characters will then be printed beginning in that position. As in other functions, the X and Y arguments may be expressions. Terminals are assumed to begin with row 0 (top of screen) and column 0 (left end of each row).

If the value of X is negative the function is interpreted as a special terminal command and the command code must be specified as the Y argument. The codes will be transmitted to the terminal driver (TDV module in 1,6) which will do the actual interpretation and perform the special function. The following list will give the standard codes in use for the ADM3, SOROC and Hazeltime CRT terminal drivers:

Code	Function
0	Clear screen
1	Cursor home (upper left corner)
2	Cursor return (column 0 without line-feed)
3	Cursor up one row
4	Cursor down one row
5	Cursor left one column
6	Cursor right one column
7	Lock keyboard
8	Unlock keyboard
9	Erase to end of line
10	Erase to end of screen
11	Protect field (reduced intensity)
12	Unprotect field (normal intensity)
13	Enable protected fields
14	Disable protected fields
15	Delete line
16	Insert line

The actual routines that perform the screen controls are in the specific terminal drivers and not in AlphaBasic itself. Therefore, if you have a different terminal and you write a driver to perform the above functions for the terminal in use it will operate properly with AlphaBasic with no modifications to either the compiler or to your programs. Note that most terminals do not support all of the above commands. Commands that are not supported on the terminal in use will merely be ignored by the driver.

FORMATTING NUMERIC DATA VIA THE "USING" MODIFIER

Often it is desirable to format a numeric value without having to immediately print it on the terminal or output it to a file. The PRINT USING performs the formatting but the output must be to the terminal or a file which can cause extra code to be generated. Formatted numeric data is handy for creating print image lines and headers. Some more exotic operations would allow pre-inspection of numeric data and string manipulation of that data for specialized applications. AlphaBasic allows formatting numeric data into its string equivalent with the USING expression modifier. Basically the format for doing this is:

```
A$ = B USING C$
```

The formatting of the numeric value in B is performed using the format mask in C$ and the result is left in A$ as a string. The mask must be a string and the formatting rules are the same as those used for the PRINT USING statement. The actual syntax rule delivers a string result anytime a numeric expression is followed by the word USING and the corresponding mask string. The following complex statement is legal:

```
A$ = "ABCD" + ((B*HOURS) USING (B$[2,4]+"####")) + "END"
```

Although it is academic in nature, a restriction exists in that the USING modifier is not recursive; i.e. the mask string itself cannot be the result of a USING modified expression.

PROCESSING INDEXED SEQUENTIAL FILES

AlphaBasic has the ability to process indexed sequential files by linking to the ISAM assembly language package which must reside wither in system memory or individual user memory. Multiple directory files are supported via some elementary ISAM statements which allow the direct control of index file and data file items. The ISAM package as implemented requires more direct user control over the files than other implementations for two main reasons. The major reason is memory limitations which restrict the amount of "smarts" that could be put into the AlphaBasic runtime package for handling ISAM files. The second reason is the current structure of the existing ISAM package which precluded a more extensive implementation within the timespan allocated to the project. We feel that the version we are releasing will provide sufficient control for all users even though the initial training and programming may at first appear more complex than other versions of ISAM handlers. For more detailed information on ISAM files and the ISAM assembly language package the user is referred to the separate manual titled "ISAM System User's Guide".

FILE STRUCTURE

An indexed sequential file consists of one data file and one or more index files which link to the data file. The data file is structured identical to a normal random access file with the additional restriction that all records which are not currently active are linked to each other in a chain called the "free data record list". All data records reside in the data file and the data records may be any size up to the maximum of 512 bytes. As in the normal random access file, data records will not be split across physical 512-byte block boundaries in the file. Index files are arranged in a complex balanced tree structure and contain one symbolic key for each active data record plus a link to that data record in the data file. This link is the relative record number and is used in the same manner as its counterpart in a normal random access file. The index file also contains an array of internal links which comprise the sequential access tree structure.

Two references used in this manual may be confusing if they are not understood. References to an "indexed file" are made when speaking of the entire file structure in general including the data file and one or index files. References to an "index file" are used when specifically speaking of the portion of the structure which contains only the symbolic keys and the tree links. Index files may be primary or secondary and have the extension IDX.

All indexed sequential files must be created by the ISMBLD program prior to access by any AlphaBasic program. There is no method for the creation of a new indexed file within the AlphaBasic language since this would require a prohibitive amount of seldom-used code for execution. The user may, however, create indexed files in a system structure by using the feature that allows a BASIC program to create and then execute a command file. This command file could set up parameters and then call the ISMBLD program to perform the actual creation of the files.

For compatibility with existing structures the data file must have an extension of IDA and all index files must have an extension of IDX. There must be at

least one index file which is called the primary index file. There may also be additional index files called secondary index files which also link to the primary data file. The primary index file must always be opened in any program in order to gain access to the data file. This is true even if you only intend to access the data file through one of the secondary index files in the current program. For information on file structures and operating the ISMBLD program refer to the ISAM System User's Guide.

SYMBOLIC AND RELATIVE KEYS

Indexed files are accessed by one of two specific types of keys. The relative key is already familiar to us since it is the same type of key used to access normal random files. A relative key when used with an indexed file is used only to access a specific record in a data file. The relative key must be a specific floating point variable which is specified in the OPEN statement. The symbolic key is new to us and is used only with indexed files. Symbolic keys are ASCII strings of variable lengths and are used to access the index file (primary or secondary). Symbolic keys are specified in the ISAM statements when accessing the index file and are used to retrieve the relative key of the associated data record in the data file. The concept of symbolic verses relative keys and their different uses is an important one and misuse of them will cause the ISAM system to malfunction in a number of ways. Symbolic keys are used with the ISAM statement and relative keys are used with the READ and WRITE statements. In most instances the use of the relative key will be transparent to the user and will merely be a device automatically set up and referenced by the above calls.

THE ISAM STATEMENT

Indexed files are accessed by a special statement in AlphaBasic called the ISAM statement. This statement has the general form:

ISAM #<file>, <code>, <symbolic-key>

All ISAM statements follow the above format using a different numeric value in <code> to specify the specific function to be performed by the ISAM package. All ISAM statements directly translate into a specific type of call to the assembly language ISAM program which must be in memory as explained previously. A symbolic key must always be specified even for those functions which do not require the use of one (this simplifies syntax checking and execution coding). A dummy string variable may be used if desired. Briefly, the following codes are used by the ISAM statement:

1 - Find a record by symbolic key
2 - Find the next sequential record in file
3 - Add a symbolic key to an index file
4 - Delete a symbolic key from an index file
5 - Locate next free data record in data file
6 - Delete a record from data file and return to free list

An error will result if an ISAM statement is executed with the value of <code> not equal to one of the above valid numbers. The code may be any legal numeric

expression which is resolved at runtime.

OPENING AN INDEXED FILE

As with other types of files, an indexed file must be opened with a specific file number prior to any references to the file by other statements. The OPEN statement follows the same format as that used by the normal random files.

```
OPEN #<file>, <filename>, INDEXED, <record-size>, <relative-key>
```

The filename must refer to the name given to the index file during the ISMBLD creation. If this is a call to open a secondary index file the user must have already previously opened the corresponding primary index file on another file number so that the data file may be accessed.

As an example, assume that an indexed file structure consists of the primary index and data files named MASTER.IDX and MASTER.IDA respectively. The structure also has secondary index files named ADRESS.IDX and PAYROL.IDX which access the MASTER.IDA file in different sequences. If it is desired to process the file structure via the sequence used by the ADRESS.IDX index file the following two statements would be required:

```
OPEN #1, "MASTER", INDEXED, RECSIZ, RELKEY
OPEN #2, "ADRESS", INDEXED, RECSIZ, RELKEY
```

Note that the record size expression (RECSIZ) and the relative key variable (RELKEY) are identical in both statements. This is important since they both refer to the same data file (MASTER.IDA). ISAM statements may then be made referring to either index file (#1 or #2) but all READ and WRITE statements must be made to the data file which is associated with the primary index file (#1). In other words, READ and WRITE statements must not be made to file #2.

ISAM STATEMENTS

There are six functions which are performed by the ISAM statement with the codes 1-6 as listed previously. They will be explained in more detail in this section. In the following descriptions there are codes which either require a relative key as input or return a relative key to be used when accessing the data record. In all cases this relative key will be retuned in the variable specified in the OPEN statement for the index file being accessed by the ISAM statement. This then leaves the system properly set up for an immediate access to the corresponding data record via a READ or WRITE statement.

Code 1 - the specified index file is searched for the key which matches the symbolic key in the statement. If a match is found the associated relative key will be delivered back for access to the data file. If the key is not found an error code 33 will be returned (see section on error processing).

Code 2 - the specified index file is accessed and the next sequential key is located. The corresponding relative key is returned in preparation for a READ or WRITE to the data file. If this is the first access to the file following the OPEN statement, the first sequential key will be located. If this statement

follows a previous code 1 statement, the next sequential key following the code 1 key will be located. If there are no more keys in the index file and end-of-index-file error (38) will be returned and no further acceses should be made to the data file until another ISAM call is made which returns a valid relative key.

Code 3 - the specified symbolic key is added to the index file along with the relative key which must be in the corresponding variable specified in the OPEN statement. This relative key will normally be already set up by a prior code 5 ISAM statement which delivered the next free data record to be used. This relative key then becomes the result of any index search which locates this specific associated symbolic key.

Code 4 - the specified symbolic key is located in the index file and deleted from it. The corresponding data record relative key is returned so that the data record may then be deleted and returned to the free list by using a code 6 ISAM statement. If the symbolic key is not located in the index file a record not found error will be returned.

Code 5 - the next available data record is extracted from the free list and the relative key is returned in preparation for a code 2 index key addition statement. If no more data records are free in the data file a data file full error will be returned. All free records in the data file are kept in a linked list called the free list. This list is built initially by ISMBLD and contains all the records in the data file. As code 5 ISAM statements are executed the free list delivers these records and as code 6 ISAM statements are executed the records are again returned to the free list for reuse. The index file is not modified and the symbolic key in the statement is ignored. This call must be made only to the primary index file number.

Code 6 - the data record specified by the relative key is returned to the free list for reuse by a code 5 call. The index file is not modified and the symbolic key in the statement is ignored. This call must be made only to the primary index file number.

READ AND WRITE STATEMENTS

The ISAM calls do not access the data records themselves but merely deliver back the relative key of the associated data record to be used. Normal READ and WRITE statements are then used to actually retrieve or write into the data record itself. These READ and WRITE statements follow the same format used when accessing a normal random access data file in AlphaBasic. The relative key associated with the primary file (as specified in the OPEN statement) must contain a valid relative key for the operation or an error will result. READ and WRITE statements as mentioned before must only be made using the primary index file number.

CLOSING AN INDEXED FILE

In order to insure that all data records have been rewritten to the data file and that all links in the index file have been properly updated and rewritten to the disk it is imperative that all index files (primary and secondary) be closed

using the normal CLOSE statement and referencing the correct file number. Failure to do so may result in the link structure being sacrificed to the god TRON (god of electrons and integrated circuits).

CREATING AN INDEXED FILE

The steps used in the initial creation of an indexed file will be traced here. Initially, the structure must be created using the ISMBLD program. Refer to the ISAM System User's Guide for details on this procedure. Any secondary index files must also be created by the ISMBLD program. The program to add the data items will then open all index files associated with the structure.

For each new data record to be added, the following steps are performed. The next free data record is retrieved (ISAM code 5) and the data record is written into it with a WRITE statement. One symbolic key is then added to each of the index files using ISAM code 3 statements. All keys will therefore link to the same data record.

After all data records have been written to the file, all files are closed.

READING AN INDEXED FILE SEQUENTIALLY

To read an indexed file in the sequence of the symbolic keys the file is first opened by the program. If it is desired to read in the sequence of a secondary file both the primary and secondary files must be opened.

Each record is retrieved by executing first an ISAM code 2 statement followed by a READ statement. Remember that the READ statement must be to the primary file even though a secondary index file is being used for the sequential accesses. After each ISAM code 2 statement the user should check for an end-of-file condition using the ERF(X) function to determine when no more data is left. Refer to the section on error processing.

Close all files.

READING AN INDEXED FILE RANDOMLY BY KEY

To read a file randomly by symbolic keys the files are opened as above for sequential access. As many secondary index files may be opened simultaneously as will be required for the random mode processing.

Each data record is located with an ISAM code 1 statement giving the symbolic key and the file number of the index file to which the symbolic key pertains. Acheck should be made for a record not found error at this point indicating that the symbolic key was not located in the specified index file. Assuming the key is valid a READ statement should be made to the primary file to get the correct data record. This operation may be performed for each data record to be read.

Close all files.

UPDATING DATA RECORDS

Data records may be updated by locating the data record via either method above and then updating the data buffer with the new data desired. This is followed by a WRITE statement to rewrite the data record. The WRITE statement must be made to the primary file. The index files are not altered in this operation.

Note that the above method should only be used to alter data that is not a direct part of any symbolic key. To change a symbolic key you must delete the key in the correct index file and then add the new key with another ISAM statement. The data record need not be deleted and recreated during this operation unless necessary for complete new data.

DELETING A DATA RECORD

The deletion of a data record in an indexed file structure involves not only the deleting of the data record itself but also the deleting of all symbolic keys associated with that data record. All index files must be open for this operation. The data record is first located by one of the symbolic keys (via ISAM code 1) and then the data record is read into the buffer with a READ statement to the primary file. Each symbolic key must then be extracted from the data record and used to delete each key from its associated index file with successive ISAM code 4 statements. The data record itself is then deleted and returned to the free list with an ISAM code 6 statement.

NOTE: A good check on the structure would be to store the relative key in another variable and then compare the relative keys returned by each ISAM code 4 statement to insure that the symbolic keys all did indeed link to the correct data record. You should also check each ISAM statement for any possible error that might otherwise go unnoticed.

ERROR PROCESSING

Every ISAM statement executed may potentially result in some form of error. Errors will fall into one of two categories: hard or soft. Hard errors are defined as errors which are returned to the ISAM processor from the monitor file service system indicating some invalid disk operation. Soft errors are those which occur within the ISAM processor indicating an error or condition peculiar to ISAM files only.

AlphaBasic handles these two types of errors in different ways. Hard errors will cause the standard BASIC error processor to be invoked resulting in either a message and program abort or an error trap if ON ERROR GOTO is in effect. These errors may be detected with the normal error processing defined in the section dealing with the ON ERROR GOTO statement.

Soft errors will never result in an error message or error trap and it is therefore up to the programmer to test for these error conditions after every ISAM statement. This is done by using the ERF(X) file error function where X is the file number used in the ISAM statement. The ERF function operates in a similar fashion to the EOF(X) function. If the ERF function returns a zero the preceding ISAM operation was successful. If the value returned is not zero then

an error or abnormal condition was detected and proper corrective action should be taken in the program prior to the next access to the file.

Current soft error codes in effect are:

32 - illegal ISAM statement code
33 - record not found in index file search
34 - duplicate key found in index file during attempted key addition
35 - link structure is smashed and must be recreated
36 - index file is full
37 - data file is full (free list is empty)
38 - end of file during sequential key read

CHAINING TO OTHER PROGRAMS AND SYSTEMS

AlphaBasic supports the CHAIN statement which terminates execution of the current program and initiates the execution of a new program or system function. The new program to be executed must be named in the CHAIN statement itself and may be a full file specification if desired. The program named in the statement may be another AlphaBasic program (compiled only) or it may be a system command or command file name. This allows a program to execute a command file and invoke system commands if required.

CHAINING TO ANOTHER ALPHABASIC PROGRAM

The default extension of the file specification in the CHAIN statement is RUN which names a new AlphaBasic program to be executed. If the extension of the evaluated file specification is indeed RUN (either explicitly or by default) the new program is loaded into memory and executed. All variables in the new program are first cleared to zero prior to execution. The program must be compiled and must be in the current user area on disk unless an explicit area is named in the file spec. The program may also optionally be resident in user or system memory if desired. Some examples of legal statements are:

```
CHAIN "PAYROL"
CHAIN "PAYROL.RUN"
CHAIN "DSK1:PAYROL[101,13]"
```

Due to the fact that programs are compiled and not interpreted there is no means for executing a program at any entry point other than its physical beginning. There is also no internal method for passing parameters between programs (sometimes referred to as "common" area) but this can be easily accomplished in a number of ways by making use of the XCALL statement and creating a common area within an external subroutine. A parameter could also be passed to the next program using this method which is then used in a computed GOTO statement to effectively begin execution at one of several points in the new program based on the value passed in the parameter.

CHAINING TO SYSTEM FUNCTIONS

It is sometimes desirable to transfer execution to a system function or a command file from a BASIC program. If the name of the file in the CHAIN statement does not have the RUN extension it is assumed to be a system command function. In this case the AlphaBasic runtime package will create a dummy command file at the top of the current user partition and then transfer control to the monitor command processor. The monitor will then interpret this dummy command file as a direct command and will continue execution at that point. Note that the dummy command file created by the runtime package is merely the one-line name specified in the CHAIN statement and not the command file itself which may be the target function desired. Some valid examples are:

```
CHAIN "SYSTAT.PRG[1,4]"
CHAIN "TEST1.CMD"
CHAIN "DSK0:MUMBLE.CMD[2,2]"
```

Note that if the account number is not specified the action taken will be the same as if the command was entered directly from the keyboard. In other words, programs and command files will normally be searched for in the user area only due to the fact that the extension had to entered explicitly. Note also that the system function will be executed as a mainline function and not as a subroutine to the runtime system. This means that if you wish to automatically return to some AlphaBasic program you will have to execute a command file whose final command is a RUN command to begin the execution of said AlphaBasic program. Confused?? Me too!! Good luck.

July 1979
DWM-00100-44

ADDENDUM TO THE ALPHABASIC USER'S MANUAL

1.0 INTRODUCTION

The purpose of this document is to provide additional information for the BASIC programmer until such time as we can issue a new AlphaBASIC user's manual (part number DWM-00100-01). For more information on using the AlphaBASIC system, turn to the "AlphaBASIC User's Manual."

1.1 Contents

The next few sections discuss new AlphaBASIC features that are not discussed in the current AlphaBASIC manual. Section 10.0 lists all messages displayed by the AlphaBASIC system, and Section 11.0 lists all reserved keywords used by AlphaBASIC.

2.0 EDITING MASKS

The section in the current BASIC manual that discusses formatted output describes the use of the PRINT USING and USING statements which allow you to format output into specific character positions by use of editing masks.

In addition to the masks mentioned, there exists one type of mask that you can use to generate a number with leading zeros. This mask takes the form of one standard numeric mask character, #, followed by a series of Zs. The total size of the output string will be the number of Zs plus the one #. For example:

```
PRINT 123 USING "#ZZZZZ"
```

yields:

```
000123
```

3.0 FILEBASE

During normal operation, the first record in a random file is referred to as record number zero (i.e., you set the record number variable to zero to access the first record in the file). In some applications it is desirable to have this first record referred to by some number other than zero. This is often done to allow you to use zero to flag some special condition, such

as a deleted record. The FILEBASE command allows you to set the number used to refer to the first record to any value. For example:

```
FILEBASE 1
```

tells BASIC that the first record in the file is record number one, not record number zero. You may use any numeric argument with FILEBASE.

Note that FILEBASE does not associate its value with a file, but is only in effect in the program where it is executed. If one program uses a FILEBASE command when referencing a file, all other programs which reference that file should also use a FILEBASE command.

4.0 EXTENDED TAB FUNCTIONS

Contrary to the statement in the AlphaBASIC manual, the home position of the cursor (the upper left-hand corner) is 1,1 NOT 0,0.

In addition to the standard TAB(-1,n) functions listed in the manual, the following are also available:

Code	Function
17	Delete Character
18	Insert Character
19	Read Cursor Address
20	Read Character at Current Cursor Address
21	Start Blinking Field
22	End Blinking Field
23	Start Line Drawing Mode (enable alternate character set)
24	End Line Drawing Mode (disable alternate character set)
25	Set Horizontal Position
26	Set Vertical Position
27	Set Terminal Attributes

Not all terminal drivers have all of the functions above simply because all terminals are not able to perform all of these functions. If your terminal has additional features, Alpha Micro recommends starting at 64 (decimal) when you assign function codes in your terminal driver.

5.0 MAP STATEMENTS

As of BASIC version 4.0, all MAP statements must appear at the front of a program before any executable code.

6.0 LIBRARY SEARCHING

Whenever a program (called via RUN or CHAIN) or a subroutine (called via XCALL) is requested, BASIC follows a specific pattern in looking for the requested module. If you specify a PPN, then BASIC uses the current default device and the specified PPN. If you specify no PPN, the search sequence is as follows:

```
Default disk:[User P,PN]
Default disk:[User P,0]
DSK0:[7,6]
```

Note that earlier versions of BASIC (pre-4.2) used a different search algorithm that was in reverse of the one outlined above.

7.0 AUTOMATIC SUBROUTINE LOADING

When a BASIC program calls a subroutine via an XCALL statement, BASIC attempts to locate the subroutine in user or system memory. If it is unable to do so, it attempts to load the subroutine off the disk, following the search pattern outlined above. If a BASIC fetches a subroutine from disk, BASIC loads it into memory only for the duration of its execution. Once the subroutine has completed its execution, it is removed from memory. Therefore, if a subroutine is to be called a large number of times, it is wise to load it into memory to avoid the overhead of fetching the subroutine from disk.

8.0 ADDITIONAL ERROR MESSAGES

In addition to the error codes defined on page 56 of the AlphaBASIC manual, two more error codes exist. We give a complete list of all BASIC messages in Section 9.0.

```
32      Invalid filename
33      Stack overflow
```

9.0 DISK COMPILER PROGRAM (COMPIL)

To enable you to compile programs that are too large to fit into memory, we provide a disk-based compiler (COMPIL). COMPIL is a two-pass compiler that gains memory space by omitting the interactive features of BASIC. COMPIL produces .RUN modules that are completely compatible with those produced by the interactive system.

9.1 COMPIL Operation

To use COMPIL, enter:

```
.COMPIL filespec{/switches}
```

where filespec selects the .BAS file you want to compile, and {/switches} select one or both of the optional COMPIL options. COMPIL allows the compilation of source files located on any device or in any account. COMPIL always places the resultant .RUN module in the account and device you are currently logged into (for BASIC versions 4.2 and later). BASIC does not support wildcarded filespecs.

9.1.1 Operation Switches

COMPIL allows the use of two switches, /O and /T. The /O switch is the same as the /O switch in the interactive compiler. That is, it tells COMPIL to omit line number references from the compiled code. This reduces the total object code size, but it prevents COMPIL error messages from reporting the line numbers where errors occurred.

The /T switch is primarily designed for debugging purposes. If you specify the /T switch, as COMPIL scans each source line of your program it displays that line on your terminal. If a problem occurs during the compilation, you can use the /T switch to determine the line in which the problem is occurring. You may also use this switch to gauge the compilation speed of various statements.

9.2 Compiler Messages

COMPIL reports a number of statistics on your terminal as it compiles. A typical compilation might look something like this:

```
.COMPIL ACMSLS
Phase 1 - Initial work memory is 2310 bytes
Phase 2 - Adjust object file and process errors
Illegal MAP level - 350 MAP FILL'7,S,2
Syntax error - 980 SLSMTD = SLSMTD;SSLAMT
Memory usage:
        Total work space - 4712 bytes
        Label symbol tree - 322 bytes
        Variable symbol tree - 1186 bytes
        Data statement pool - 0 bytes
        Variable indexing area - 274 bytes
        Compiler work stack - 140 bytes
        Excess available memory - 11918 bytes
```

Note that any error messages are reported during pass two, and that the source line containing the error is typed on your terminal. The "Excess available memory" line is useful for letting you know how close you are to running out of memory. If you do run out of memory during a compilation,

you see the message "[Out of memory - compilation aborted]", and COMPIL returns your terminal to AMOS command level.

9.3 Line Numbers

Because AlphaBASIC allows the use of labels, and because COMPIL assumes that you are using one of the text editors (VUE or EDIT) to maintain your source code, line numbers are optional in source code given to COMPIL. By omitting line numbers, and with judicious use of indentation, you can give source code a much more structured look than is normally possible in BASIC.

9.4 Continuation Lines

COMPIL allows the use of continuation lines within the source program. This is especially useful for giving source code a structured appearance. Specify a continuation line by making an ampersand (&) the last character on that line. For example:

```
10      IF (X < 12.2) OR (B > 0) THEN &
                J = X/167.2 &
            ELSE &
                J = B
20      Q = 1252
```

The maximum size of any line, including any continuation lines, is 500 characters.

If a program with continuation lines is loaded into the interactive compiler, the lines are concatenated into one line. Therefore, loading and saving a program with continuation lines under the interactive compiler (BASIC) results in the elimination of the continuation lines.

10.0 MESSAGES OUTPUT BY THE ALPHABASIC SYSTEM

Below is a complete list of all messages output by the AlphaBASIC system (i.e., BASIC, RUN, and COMPIL), along with a brief explanantion of each message.

Bitmap kaput
: Your program attempted a file operation (OPEN, ALLOCATE, etc.) on a device with a bad bitmap.

Break at line n
: The program reached the breakpoint that was set at line n.

COMPILE
: BASIC is telling you that it is compiling your program.

Can't continue
You have attempted to continue a program which is not stopped at a breakpoint, or which has reached a point where it can go no further (e.g., it has reached an END statement).

Cannot find xxxxxx
The program xxxxxx was not found.

Compile time was x.x seconds
BASIC is telling you how long (in elapsed time, not compute time) it took to compile your program.

DELETE what?
You have specified a DELETE command without specifying what line(s) are to be deleted.

Device does not exist
The device you specified in a file operation (OPEN, LOOKUP, etc.) does not exist.

?Device driver must be loaded into user or system memory
If you are accessing a non-DSK device, the appropriate device driver must be loaded into user or system memory.

Device error
An error has occurred on the referenced device.

Device full
The specified device has run out of room during a WRITE, CLOSE, or ALLOCATE operation. Remember that an ALLOCATE requires contiguous disk space, so that a Device full error may occur when there are still a number of non-contiguous blocks available.

Device in use
The specified device is currently assigned to another user.

Device not ready
The specified device is not ready for use.

Disk not mounted
The specified disk has not been mounted. Mount it via the MOUNT monitor command or via the XMOUNT subroutine.

Divide by zero
Your program attempted to perform a division by zero.

Duplicate label
Your program has defined the same label name more than once.

*** End of Program ***
You have reached the end of the program during single-stepping.

Enter <CR> to continue:
You have reached a STOP statement in your program. You may continue from the STOP statement via a carriage-return, or may abort the run via a Control-C.

File already exists
Your program tried to create a file which already exists.

File already open
You have attempted to open a file that is already open on the same file number.

File not found
BASIC was unable to locate the specified file.

File spec error
The file specification you gave in a file operation (OPEN, LOOKUP, etc.) is in error. All file specifications must conform to the system standard (i.e. devn:file.ext[p,pn]).

File type mismatch
Your program tried to perform a sequential operation on a random file or vice-versa.

Floating point overflow
A floating point overflow occurred during a calculation.

IO to unopened file
Your program tried to perform input or output to a file that is not open.

Illegal GOTO or GOSUB
The format of the GOTO or GOSUB statement is invalid.

Illegal NEXT variable
The variable used in the NEXT statement is not valid (e.g., not floating point).

Illegal PRINT USING format
The edit format used in a PRINT USING statement is invalid.

Illegal SCALE argument
The argument given in a SCALE statement is invalid (the argument must range between -30 and +30).

Illegal STRSIZ argument
The argument given in a STRSIZ statement is invalid.

Illegal TAB format
Your program has incorrectly specified a TAB function.

Illegal expression
The specified expression is not valid.

Illegal function value
The specified function value is not valid for the particular function.

Illegal line number
The specified line number is invalid (e.g., not between 1 and 65534).

Illegal or undefined variable in overlay
The variable specified in a MAP statement overlay (via @) has not been previously defined, or is not a mapped variable.

Illegal record number
The relative record number specified in a random file processing statement (i.e., READ or WRITE) is either less than the current FILEBASE or outside of the file.

Illegal size for variable type
The specified variable size is not valid for the particular variable type. Floating point variables must be size 6, and binary variables must have size 1-5.

Illegal subroutine name
The name specified as a subroutine is not valid.

Illegal subscript
The subscript expression is not valid.

Illegal type code
The variable type code specified in a MAP statement is not one of the valid types.

Illegal user code
The specified PPN was not found on the specified device, or is not in a valid format.

Insufficient memory to load program xxxxxx
The RUN program did not find enough free memory to be able to load the specified program.

Invalid filename
The specified filename was not a legal filename.

[Invalid syntax code]
An internal error has occurred in BASIC. Please notify Alpha Micro of this error. Provide an example of what caused it.

Line number must be from 1-65534
The line number entered was not in the range of legal line numbers.

Line x not found
The specified line was not found for a DELETE, LIST, etc., operation.

NEXT without FOR
A NEXT statment was encountered without a matching FOR statement.

No breakpoints set
BASIC is telling you that there are currently no breakpoints set in your program.

No source program in text buffer
You tried to compile when there was no program in memory.

Operator interrupt
You typed a Control-C to interrupt program execution.

Out of data
A READ statement was encountered after the data in all DATA statements had been used.

Out of memory
BASIC is telling you that it has run out of memory in which to execute your program.

Out of memory - Compilation aborted
COMPIL is telling you that it does not have enough free memory to finish compiling your program.

Program name:
You tried to SAVE or LOAD a program without providing a filename. Enter the filename at this point.

Protection violation
Your program tried to write into another account where you do not have write priviliges.

RESUME without error
A RESUME statement was encountered, but no error has occurred.

RETURN without GOSUB
A RETURN statement was encountered, but no corresponding GOSUB has been executed.

Record size overflow
Your program tried to read a file record into a variable larger than the file record size.

Redimensioned array
You tried to redimension an array.

Runtime was x.x seconds
BASIC is telling you how long it took to run your program.

?Runtime package (RUN.PRG) not found
BASIC or COMPIL was unable to locate the runtime package, or did not have sufficient memory in which to load it.

Stack overflow
: BASIC's internal stack has overflowed. This is most often caused by such operations as nesting GOSUBs too deep, or branching out of FOR-NEXT loops.

Subroutine not found
: The specified subroutine could not be found.

Subscript out of range
: The specified subscript is outside the range specified in the DIM or MAP statement for the subscripted variable.

Syntax error
: The syntax of the specified line is invalid.

System commands are illegal within the source program
: BASIC system commands (LOAD, DELETE, LIST, etc.) are not valid within a BASIC source program.

System error
: A system error has occurred during the execution of the specified line. System error is used as a catch-all error message for a variety of unlikely occurrences.

Temporarily all arrays must be less than 32K
: The array size you specified is larger than 32K bytes.

Undefined line number or label
: The line number or label specified in a GOTO or GOSUB statement is not defined within the program.

Write protected
: Your program tried to write on a write-protected device.

Wrong number of subscripts
: The number of subscripts specified is not the same as the number defined in the DIM or MAP statement for the subscripted variable.

11.0 RESERVED WORDS

Below is a list of the reserved words used by the BASIC compilers (BASIC and COMPIL). You MUST not use any of these reserved words as variable names or labels.

Reserved Word	Meaning
ABS	absolute value
ACS	arccosine
ALLOCATE	allocate file
AND	logical AND
ASC	ASCII value
ASN	arcsine
ATN	arctangent
BREAK	set breakpoint
BYE	exit to monitor
BYTE	memory byte
CALL	call subroutine
CHAIN	chain next program
CHR	character value
CHR$	character value
CLOSE	close file
COMPILE	compile program
CONT	continue execution
COS	cosine
DATA	data statement
DATE	system date
DATN	double arctangent
DEF	define function
DELETE	delete lines
DIM	dimension
ELSE	else
END	end of program
EOF	end of file
EQV	logical equivalence
ERF	file error
ERR	error status
ERROR	error
EXP	exponentiation
EXPAND	expand mode on
FACT	factorial
FILEBASE	file base offset
FIX	fix
FOR	loop initiation
GO	program jump
GOSUB	call subroutine
GOTO	program jump
IF	conditional test
INDEXED	indexed
INPUT	input data
INSTR	search string
INT	integer

IO	input/output
ISAM	ISAM control
KILL	kill file
LCS	lower case string
LEFT	left string
LEFT$	left string
LEN	length string
LET	variable assignment
LINE	line
LIST	list text
LOAD	load program
LOG	natural logarithm
LOG10	base 10 logarithm
LOOKUP	lookup file
MAP	map variable
MAX	maximum value
MEM	memory size
MID	mid string
MID$	mid string
MIN	minimum value
NEW	new program
NEXT	loop termination
NOEXPAND	expand mode off
NOT	logical complement
ON	on (goto,gosub,error)
OPEN	open file
OR	logical OR
OUTPUT	output
PRINT	print on terminal/file
RANDOM	random
RANDOMIZE	randomize RND function
READ	read data
REM	remark line
RESTORE	restore data
RESUME	resume after error
RETURN	subroutine exit
RIGHT	right string
RIGHT$	right string
RND	random number
RUN	run program
SAVE	save program
SCALE	set scale factor
SGN	sign
SIGNIFICANCE	set significance
SIN	sine
SPACE	spaces
SPACE$	spaces
SQR	square root
STEP	step
STOP	stop program
STR	numeric to string conversion
STR$	numeric to string conversion
STRSIZ	set string size

SUB	sub (gosub)
TAB	tab
TAN	tangent
THEN	optional statement verb
TIME	system time
TO	to
UCS	upper case string
USING	using
VAL	string to numeric conversion
WORD	memory word
WRITE	write file
XCALL	external subroutine call
XOR	logical XOR