A/BASIC INTERPRETER REFERENCE MANUAL

SOFTWARE LICENSE AND LIMITED WARRANTY:

A/BASIC is copyrighted property of Microware Systems Corporation and i furnished to its customers for use on a single computer system owned b the customer. The program may not be copied in any form except for us on the customer's computer system. This license is not transferable a the customer may not make this software available to any third party without express written consent of Microware Systems Corporation.

Purchase of this program and manual is considered implied consent of t provisions of the software license and warranty.

Microware Systems Corporation warrants this software to be free from defects for a period of 90 days after date of purchase. Any correctio may be made by means of printed or magnetic media at the option of Microware Systems Corporation.

Microware Systems Corporation reserves the right to make changes withc notice (except within the warranty period) in the interest of product improvement at any time. Microware Systems Corporation cannot be responsible for any damages, including indirect or consequential, caus by reliance on this product's performance or accuracy of its documenta

---

MICROWARE SYSTEMS CORPORATION
P.O. BOX 4865
DES MOINES, IOWA 50304
(515) 265-6121

FIRST EDITION

PART NUMBER BASI-M

INTRODUCTION

The A/BASIC Interpreter is a fast BASIC Interpreter for 6800 family microcomputers. Though it was primarily created as an interactive edit/debug companion to Microware's A/BASIC Compiler, it can be used alone in many applications where speed and versatility rule out use of other types of interpreter.

Users are cautioned that interpreters and compilers have fundamental differences that preclude 100% compatibility, so A/BASIC Interpreter programs should be checked against the compiler's requirements before compilation.

The source listing to the Interpreters I/O package is also provided for users who wish to modify it for special purposes. We cannot, however, be responsible for results or performance of this software when so modified.

This program is supplied to operate with the disk or tape configuration identified by the version number, and will run correctly on system hardware commonly used with the particular operating system or monitor. For disk version a file called "INFO" is included on the disk which contains information for the specific version.

GETTING STARTED

For disk systems, the interpreter is supplied as a command file called "BASINT", and should automatically load and start when this command is entered. Cassette versions are supplied on a Kansas City Standard 300 baud tape in Motorola "S1" hex format.

All versions except MDOS have a cold-start (reset) jump at $0100 and a warm start at $0103. For MDOS these addresses are $2000 and $2003, respectively.

INPUT/OUTPUT

The following control characters are used throughout:

CONTROL X = delete entire line
CONTROL H = backspace
CONTROL C = program interrupt (see below)

Any of the above may be altered to suit any particular system by changing the corresponding entry in the I/O table (source listing supplied). The control-C interrupt function is not implemented in most versions because of its highly system-dependent nature. This function requires that the interface control register for the system keyboard to be sampled regularly. The user may install this small routine if desired. Specific information can be found in the I/O source listing.

BASIC COMMANDS

These commands cannot be used as program statements.

NEW

Clears memory and resets the interpreter. The previous program is destroyed.

SIZE

Causes the interpreter to print the program size (number of bytes).

SAVE

Outputs a copy of the program in source text format to tape or disk. In disk systems the interpreter will respond with:

FILE NAME:

to prompt the user to enter the name of the file to be created. The file name format must agree with the disk operating system's requirments and syntax. Any error message displayed will directly correspond the DOS error reporting conventions.

The output file can be used as source for the A/BASIC compiler, or can be modified by the system's text editor.

LOAD

Causes a source program to be read into the interpreter from disk or tape. Disk versions will print the prompt:

FILE NAME:

As with the SAVE command, file names and error reporting are dependant on the particular operating system's conventions. LOAD does not clear the previous program (if any), making it possible to develop a program in segments.

RUN

Starts execution of the beginning with the lowest-number line.

BASIC COMMANDS - CONT'D

LIST

Prints one or more lines of the BASIC program. There are several variations of this command depending on the parameters given:

LIST	List the entire program.
LIST,number	List the single line specified.
LIST,number1,number2	List all lines starting with line number1 up to and including number2.
LIST,number,	List all line from number specified to the end of the program.
LIST,,number	List all lines from the beginning of the program up to and including the line number specified.

Because the A/BASIC Interpreter is an incrental compiler it uses an internal format for storage of programs, the program may not be LISTed exactly the same as originally entered. Spacing between keywords, expressions, etc., will be uniformly single spaced. This will not affect the execution of the program in any way.

AUTO

Initiates the automatic line numbering mode. When active, the interpreter will automatically supply sequential line numbers at the start of every new line. This mode can be turned off by deleting a line, then entering any other command.

This command has several variations that depend on the parameter list given:

AUTO	Begin automatic line numbering. Line numbers start where previously left off using the same increment. After a NEW command or when the Interpreter is first started up, the first line number will be 10, and the initial increment factor is set to 10.
AUTO,number	Begin auto line numbering starting at the line number specified using the previous increment factor (or 10).
AUTO,number,incr	Start auto line number at the line and using the increment factor specified.
AUTO,,incr	Resume auto line numbering starting with the line number where previously left off using the new increment factor specified.

BASIC COMMANDS - CONT'D

MON

Exits the interpreter and returns to the system monitor or disk operating system.

CONT

Continue program execution after a PAUSE statement halted the program. Execution will resume with the statement following the PAUSE statement. This command will not be performed if the program was modified since the program was stopped.

VARIABLE BINDING AND STORAGE ALLOCATION

One of the major design goals of the A/BASIC Interpreter was to provide a means to interactively prepare and debug source programs for the A/BASIC compiler. To accomplish this the interpreter must behave as closely as possible to the way the compiler does. Due to major differences in the structure of interpreters as compared to compilers, exactly identical operation is not possible (or sometimes not even desirable).

The fundamental difference is how memory space is assigned to program and variable storage. With the compiler, the programmer is in total control by including BASE and ORG statements in the program which are obeyed even if conflicts result (which are almost always very fatal errors).

On the other hand, the interpreter controls memory assignments automatically and therefore eliminates potential conflicts. Unfortunately this also eliminates many useful options the programmer has in assigning storage for variables to specific addresses.

The A/BASIC interpreter's system is a compromise between both worlds. First, the ORG statement is non-functional as the interpreter must control program storage to allow interactive editing, LISTing, etc. Also, the interpreter's internal compiled representation is not machine language and program sizes would be different than regular compiled code anyway.

The process of assigning specific memory space to variables (sometimes called "binding") is often very important, so the BASE statement is implemented in the interpreter according to the rules and restrictions listed below.

1. BASE statements may be used anywhere in a program. They become effective when "executed" (different than compiler which sequentially scans statements from beginning to end without control transfers by GOTO's, etc.).

2. The interpreter will defend itself. If an attempt is made to allocate memory used by the program or the interpreter itself, an error message will be displayed and the BASE statement ignored.

3. BASE statements may be used to cause more than one variable to share the same memory space with other variables.

If a program is stopped and statements changed, it is likely that the memory assignments may also have to be changed. Variables will then lose their previous values (are set to zero or null) and may be re-bound to new addresses.

The interpreter will print the message:

** WARNING ** VARIABLES UNBOUND

to alert the user of this occurance. Some direct execution mode statements execution will also cause this to happen.

DIRECT EXECUTION

Any A/BASIC statement may be typed in without a line number and immediately executed. This can be of great usefulness when debugging a program. The PRINT statement can be used to examine the current value of variables, and the LET statement can be used to assign values to variables. Some statements may or may not produce a useful function in this mode. For example, the NEXT statement may or may not work, depending on whether or not a FOR loop using the variable N is active.

The direct execution mode also allows immediate execution of expressions. This is in effect an implied "PRINT" statement. An example: if variable A has the value 12 and variable B has the value 3 typing:

DIRECT EXECUTION - CONT'D

```
A-B
```

will result in immediate execution and the interpreter will print the result:

```
 9
```

A simple or array variable name can be entered alone:

```
A
```

will result in the interpreter printing:

```
 12
```

WARNING: Do not enter an expression starting with a number! The interpreter will think it is a numbered line and treat it as such. Placing parentheses around such expressions will avoid this problem.

If a variable is used in direct execution that has not been referenced previously, previous variable-storage bindings may be changed and a warning message printed (see VARIABLE BINDING AND STORAGE ALLOCATION).

A/BASIC PROGRAM STRUCTURE

An A/BASIC program consists of a series of source lines. A source line must begin with a line number, which is then followed by one or more A/BASIC statements. If the source line contains more than one statement a colon : character is used to separate the statements. A source line may contain up to 80 characters.

Line numbers are decimal numbers which are up to four digits and positive. These must appear sequentially in a program and may not be duplicated. Spaces in A/BASIC statements are not required however they may be used to improve readability (except when used in string constants). The interpreter will list programs with uniform (single) spacing.

The last statement of a program is an END statement and processing ceases when END is read.

EXAMPLE:

```
100 FOR N=1 TO 10 : PRINT N: NEXT N
200 A=B/C
300 PRINT A,B,C
```

ARITHMETIC OPERATIONS

NUMBERS

A/BASIC's numeric data type is internally represented as 16 bit (2 byte) two's compliment integers. This permits a equivalent decimal range of +32767 to -32768. This data representation is quite natural to the 6800's machine instruction set which allows A/BASIC to produce extremely fast and compact machine code.

Because A/BASIC supports boolean operations, unsigned 16 bit binary numbers may also be used for many functions. The range for these are: Ø to +65535. These numbers are used for referencing memory addresses in many cases.

A/BASIC programs may include numeric constants in either decimal or hexadecimal notation. In the latter case a dollar sign must precede a hex value. Either type may have a preceding minus sign to represent a negative value or a pound sign # to represent the logical compliment (1's compliment or boolean NOT).

Examples of legal number constants:

2ØØ -5ØØØ $Ø1ØØ -$3ØØØ 12345 #1 #$5ØØØ $FFFØ #$CØF1

Examples of ILLEGAL NUMBERS:

9.99 (fractions not allowed)
1000000 (number is too large)
+20 (plus sign not allowed - positive assumed if not minus)

Because binary numbers are represented as either unsigned or 2's compliment, as well as the differences between hex and decimal notation of identical numbers, all the following number constants have a binary value which are the same:

-1_{10} $FFFF #Ø 65535

NUMERIC VARIABLES

Legal numeric variable names in A/BASIC consist of a single letter A-Z or a single letter and a digit Ø-9. The following are legal variable names:

X N R2 Z9 AØ P1

If declared in a DIM statement, numeric variables may be arrays of one or two dimensions. The maximum subscript size is 255, therefore the largest one-dimensional array has 255 elements and the largest two-dimensional array has 255*255 = 65025 elements (which is too big to actually exist within the 6800 memory space). Subscripts start with 1.

When referencing subscripted variables the subscripts may be numeric constants, variables or expressions as long as the evaluated result is a positive number from 1 to 255, otherwise an error will occur.

Examples of legal subscripting:

N(M) A(12) X2(C) Z4(N,M) H(N*(A/B),A+2) R4(N*N+M)

A/BASIC considers a simple variable with the same name as an array to be the first element of an array. For example if there is a two-dimension array A(20,40) using the variable name A without any subscript is equivalent to using A(1,1).

Each numeric variable or element of an array is assigned two bytes of RAM for storage.

ARITHMETIC OPERATORS

The five legal operators for arithmetic are:

+	ADD
-	SUBTRACT
*	MULTIPLY
/	DIVIDE
-	NEGATIVE (UNARY)

There are also four boolean operators:

&	AND
!	OR
%	EXCLUSIVE OR
#	COMPLIMENT (UNARY)

All the above operators may be mixed in arithemetic expressions. The boolean operators operate in a bit-by-bit manner across all 16 bits of the numeric value.

Expressions are evaluated in the following order:

1	FUNCTIONS
2	UNARY NEGATIVE AND NOT
3	AND, OR, EXCLUSIVE OR
4	MULTIPLY, DIVIDE
5	ADD, SUBTRACT

Parentheses may be used to alter the normal order of evaluation where required. Some legal usage of expressions:

```
A*B(N,M+4)    $200+Z    A&B!C*D/F+(H+(J*2)&$FF00)    N+A(Z)/VAL("N$)
```

ARITHMETIC FUNCTIONS

A/BASIC supports the following functions:

ABS(expr)	The absolute value of the argument.
RND or RND (expr)	Next number from the random sequence. The number will be in the range Ø to +32767. If an argument is supplied, it is evaluated and used to "seed" the random number generator (randomize it).
PEEK(expr)	Used to access a byte value at an address determined by the result of the argument.
POS	The current character position in the output buffer (print position).
SWAP(expr)	Byte swap of the result of the argument.
ERR	Returns error type of most recent error condition.
ADDR	Returns memory address assigned to variable (not in compiler).

ARITHMETIC ERRORS

Arithmetic operations may produce several types of errors which may be detected and processed. Addition and subtraction may result in a carry or borrow. Either one will result in the C bit of the MPU's condition code register being set. The ON OVR GOTO and ON NOVR GOTO statements may be used to detect this. This also permits addition and subtraction in larger representation than 16 bits. (See MULTIPLE PRECISION ARITHMETIC)

Multiplication of two 16 bit numbers may result in a product of up to four bytes long. Division attempted with a divisor of zero will also produce an error which can be detected with the ON ERROR GOTO statement.

MULTIPLE PRECISION ARITHEMTIC

Sometimes it is necessary to deal with numbers larger than the basic 2-byte A/BASIC representation. A/BASIC allows addition and subtraction of numbers of multiples of 16 bits by means of the ON OVR GOTO and ON NOVR GOTO statements. OVR means overflow (carry or borrow as represented by the MPU C bit) and NOVR means NOT OVERFLOW.

The example below shows addition and subtraction of 32-bit integers using the convertion that two variables are used to store each number: A1 and A2 are the first number with A1 being the most significant bytes; and B1 and B2 used similarly. To add A1-A2 to B1-B2 the following subroutine may be used:

```
100 A2=A2+B2 : ON NOVR GOTO 200 : REM ADD L.S. BYTES
150 A1=A1+1  : REM ADD 1 TO MS BYTES FOR CARRY
200 A1=A1+B1 : REM ADD MS BYTES
250 RETURN
```

To subtract B1-B2 from A1-A2 a similar subroutine may be used:

```
100 A2=A2-B2 : ON OVER GOTO 300 : REM SUB. LS BYTES
200 A1=A1-B1 : RETURN : REM SUB MS BYTES
300 GOSUB 200 : A1=A1-1 : RETURN : REM BORROW CASE
```

For cases where multiply, divide or even floating-point arithmetic must be used, external subroutines may be used. In such cases several compiler features and capabilities may be used to simplify the interface.

1) Use the CALL statement to call subroutines.
2) Set up conventions so values are passed to the external subroutines in certain memory addresses that have been assigned A/BASIC variable names so the A/BASIC program may easily manipulate them.
3) Use A/BASIC's string processing capabilities to full advantage in handling I/O and storage of numeric values. Floating point numbers can be passed as strings in ASCII format.

STRING OPERATIONS

A/BASIC features a complete set of string processing capabilities which allow BASIC programs to perform operations on character-oriented data. Character-type data is represented in A/BASIC in string form which is defined as variable-length sequences of characters.

String Literals

A string literal or constant consists of a series of characters enclosed in quotation marks:

```
"THIS IS A STRING LITERAL"
```

Any characters may be included in a string literal except for the ASCII characters for carriage return, null or SUB ($1A - used for end-of-file on source programs). A string literal may include up to as many characters as may fit in an A/BASIC source line. The quotes are not considered a part of the string. If a quote is to be included as part of the string two are used so the literal:

```
"AN EMBEDDED "" QUOTE"
```

is interpreted to mean the constant string:

```
AN EMBEDDED " QUOTE
```

String literals are used in string assignment statements or expressions, and in PRINT or WRITE statements.

String Variables:

A/BASIC allows string variables which may be either single strings or arrays of strings. String variable names consist of a single letter A-Z followed by a dollar sign such as A$, M$, or Z$.

String variables may be used with or without explicit declaration. If a string variable is encountered for the first time in the source program without having been previously declared in a DIM statement A/BASIC will assign 32 bytes of storage for the string. This is the maximum number of characters that may be assigned to the variable. If the assignment statement produces a result which has more characters than assigned for the variable the first N characters will be stored where N is the length of the variable storage assigned.

A string variable or array may be declared to have a size of 1 to 255 characters in length if the string is declared by a DIM statement before it is used (see DIM statement description).

If the string name is declared as an array, the maximum subscript size is 255. Legal usage of string arrays require that only one subscript (which may be an expression) be used:

```
A4(5)    N$(X+5)    X$(A+(N/2))
```

String Concatenation

The string concatenation operator + is used to join strings to form a new string or string expressions. For example:

```
"NEW "+"STRING"
```

produces the value: "NEW STRING".

Null Strings

Strings which have no characters are represented as literal as "" which represents an empty string. This is typically the initial value assigned

to a string which is to be "built up". The string assignment statement:

A$=""

is somewhat analogous to the arithmetic assignment A=Ø in the sense that both cause a variable to be assigned a defined value of "nothing". This is important because before a string variable is used in a program it has a value which is random and meaningless.

String Functions

A/BASIC includes many functions which manipulate strings or convert strings to/from other types. Some of the functions which include $ in their name produce results which are of a string type and may be used in string expressions. In the description of string functions that follow, the notation:

N or M refers to a numeric-type argument which is a constant, variable or expression

X$ or Y$ refers to an argument of string type which may be a string literal, variable or expression.

The following functions produce STRING results:

CHR$(N) returns a character which is the value of the number N in ASCII.

LEFT$(X$,N) returns the N leftmost characters of X$. For example, the function LEFT$("EXAMPLE",3) returns "EXA".

MID$(X$,N,M) returns a string which is that part of X$ beginning with its Nth character and extending for M characters. For example: the function MID$("EXAMPLE",3,4) returns "AMPL".

RIGHT$(X$,N) returns the N rightmost characters of X$. An example this function is RIGHT$("EXAMPLE",3) which produces "PLE".

STR$(N) is a function used to convert a number from numeric type to string type which is a string of characters which are decimal digits. For example STR$(1234) returns the string "1234". This function has the inverse effect of the VAL function.

TRM$(X$) is a function which removes trailing blanks from a string and is typically used after a string is read from input. For example TRM$("EXAMPLE ") returns "EXAMPLE".

Note on the above functions: if there are not enough characters in the argument to produce a full result, the characters returned will be those processed until the function "ran out" of input, or a null string, whichever is appropriate.

The following functions have string argument(s) and produce a result which if of type numeric:

ASC(X$) returns a number which is the ASCII value of the first character of the string. For example ASC("EXAMPLE") returns a value of $45 or decimal 53 which is the ASCII code for the character E. This is the inverse function of CHR$.

LEN(X$) returns the length of the string. LEN("EXAMPLE") returns a value of 7. LEN("") returns a value of Ø.

SUBSTR(X$,Y$) is a substring search function which searches for the string X$ in the string Y$. If an identical substring is found the function will return a number which is the position of the first character of the substring in the target string. If the substring is not found the function returns a value of Ø. For example, the function SUBSTR("PL","EXAMPLE) returns a value of 5. SUBSTR("EXAMPLE","NOT") returns a value of Ø.

VAL(X$) converts a string of characters for decimal digits and optionally a leading minus sign to a numeric value. This has the inverse effect of STR$. If the string argument is not a legal conversion string (it has too many, non-decimal or no digit characters) a run-time error detectable by ON ERROR GOTO occurs. For example:

VAL("1234") returns a numeric value of 1234. VAL("THREE") results in an error.

String Operations on the I/O Buffer

Commonly BASICs have limitations because of the input formatting when reading mixed data types. For example BASIC input conventions cause commas which are part of the input data to break up what are really one long string, etc. A/BASIC has a special string variable, BUF$ which is defined to be the contents of the run-time I/O buffer which may be used as any other string variable. BUF$ is 129 bytes long.

The following I/O statement forms are legal for filling or dumping the I/O buffer when used with BUF$:

```
INPUT BUF$          PRINT BUF$
READ #N,BUF$        WRITE #N,BUF$
```

Example of usage of BUF$ as a variable:

```
BUF$=MID$(BUF$+A$,N,M)
```

String Expressions

String expressions may be created using string variable names, the concatenation operator and string functions. Expressions are evaluated from left to right and the only precedence of operations involved is evaluation of function arguments performed before concatenation.

At execution, string operations are performed on data moved to the string buffer, working area 255 bytes long.

Examples of legal string expressions:

```
"CAT"
A$
A$+"DOG"
LEFT$(B$,N)
A$+RIGHT$(D$,Z)+"TH"
MID$(A$+B$,N,LEN(A$)-1)
"AA"+LEFT$(RIGHT$(TRM$(A$)+B$,Z4),X+2)+C$
```

BASE Statement

Syntax: BASE = address

Used to set or change the interpreter's internal variable storage allocation pointer to the address specified. Variables encountered for the first time in subsequent statements will be assigned storage sequentially from this address. Multiple BASE statements may be used.

If the BASE statement would cause storage to be allocated which is already used by the interpreter or the program, the statement will be ignored, and a warning message will be displayed.

See the VARIABLE BINDING AND STORAGE ALLOCATION section for more information.

NOTE: The A/BASIC compiler assigns variable storage while scanning statement lines from beginning to end. The interpreter allocates storage during initial execution of the program, following program flow. The following program would be identical on either:

```
100 BASE = $4000
200 A=1
300 BASE = $5000
400 B=1
```

This one would not because the "GOSUB" would cause storage to be allocated for C first in the interpreter only.

```
100 BASE = $4000
150 GOSUB
200 A=1
300 BASE = $5000
400 B=1 : STOP
500 C=1 : RETURN
```

DIM Statement

This statement type is used to declare arrays and optionally, other simple variables. Arrays must be declared in a DIM statement before they are referenced in the program. The DIM statement may be used to declare more than one array. Arrays may not be redefined in following DIM statements. Array subscripts have a legal range of 1 to 255.

Numeric Arrays

Numeric arrays may be declared to have one or two dimensions. Two dimensional arrays are stored in row-major order. Each element of a numeric array requires two bytes of storage. Examples of numeric array declaration:

```
DIM B(20),C(10,20),D($10,$20)
```

String Arrays

String arrays may only be one-dimensional, however, the DIM statement is also used to specify the string size (1 to 255 characters) so the declaration for a one dimensional string array will have two subscripts: the number of strings and the length of each string. A single string may be declared in the DIM statement with a length specification only. Examples:

```
DIM A$(80)       one string of 80 characters
DIM B$(16,72)    16 strings of 72 characters
```

In the two examples above, A$ is used in the program WITHOUT any subscripts because it is not an array. B$ would be used in the program with one subscript because it is a one-dimensional array. For example:

```
A$=B$(N)
B$(X+2)=A$
```

Declaring Simple Variables

Because A/BASIC allocates memory for variables as they are encountered for the first time, it is often useful to declare a variable name so it may be assigned storage at a specific time. This is often the case when it is desired to assign a variable a certain memory address. A/BASIC processes a variable declared as an array but used without subscripts in the program as the first element of the array.

Declaring Simple Variables - Cont'd

Because of this a declaration of a variable in a DIM statement with a subscript of 1 is legal but the variable may be used throughout the program without a subscript.

Example: Suppose a program is to be used to read from and write to an ACIA interface at address $8008 - $8009 and a PIA at addresses $8020 - $8023, and they are to be assigned variable names. A DIM statement at the beginning of the program may be used to assign variable names to these devices:

```
BASE=$8008      set compiler data pointer
DIM A(1)        declare ACIA as variable
BASE=$8020      reset data pointer
DIM P(1),Q(1)   declare PIA "A" and "B" registers
BASE=$0030      restore data pointer for other variables
```

The program may now refer to either the PIA or ACIA by variable name.

To access the PIA "B" registers:

```
N=Q
```

or to read the ACIA:

```
N=A
```

REMARK STATEMENT

The REM statement is used to insert comments in the BASIC source program. The first three letters must be REM. On multiple statement lines the REM statement may only be used as the last statement on the line. This statement does not affect object program size or speed.

END STATEMENT

This statement ceases execution of the program and returns control to A/BASIC command mode.

TRCON, TRCOFF STATEMENTS

These statements cause the trace mode to be turned on and off.

ASSIGNMENT STATEMENTS

Arithmetic Assignment

SYNTAX: LET var = expr
var = expr

The expression is evaluated and the result is stored in the variable which may be an array. Use of the keyword LET is optional.

POKE Assignment

SYNTAX: POKE(expr) = expr

The expression is evaluated, and the result is truncated to a single (least significant) byte value which is stored at the address determined by evaluation on the expression in parentheses.

String Assignment

SYNTAX: strvar = strexpr
LET strvar = strexpr

The string expression is evaluated and the result assigned to the string variable specified, which may be an array element. If the result of the evaluation produces a result with a longer length than the size of the result variable, the first N characters only are stored where N is the length of the result variable.

CONTROL STATEMENTS

Call Statement

Syntax: CALL <address>

Description: The CALL statement is used to directly call a machine-language subroutine at the address specified. The subroutine will return to the BASIC program if it terminates with an RTS instruction and does not disturb the return address on the stack.

Examples:	CALL $EOCC	Call subroutine at address $EOCC
	CALL 1024	Call Subroutine at decimal addr. 1024

For/Next Statement

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

Description: The FOR/NEXT uses a variable <var> as a counter while performing the loop delimited by the NEXT statement. If no step is specified, the increment value will be 1. The FOR/NEXT implementation in A/BASIC differs slightly from other BASIC due to a looping method that results in extremely fast execution and minimum length. Note the following characteristics of FOR/NEXT operation:

1. <var> must be a non-subscripted numeric variable.
2. The loop will be executed at least once regardless of the terminating value.
3. After termination of the loop, the counter value will be GREATER than the terminating value because the test and increment is at the bottom (NEXT) part of the loop.
4. FOR/NEXT loops may be exited and entered at will.
5. Up to 20 loops may be active, and all must be properly nested.
6. The initial, step, and terminating values may be positive or negative. The loop will terminate when the counter variable is greater than the terminating value.

Examples: FOR N = J+1 TO Z/4 STEP X*2
FOR A =-100 TO -10 STEP -2

Gosub/Return Statements

Syntax: GOSUB <line #>
RETURN

Description: The GOSUB statement call a subroutine starting at the line number specified. If no such line exists, an error will result. The RETURN statement terminates the subroutine and returns to the line following the calling GOSUB. Subroutines may have multiple entry and return points. Subroutines may be nested to a depth of 16.

If/Then Statement

Syntax: IF <expr> <relation> <expr> THEN <line #>
IF <expr> <relation> <expr> GOSUB <line #>

Description: The IF/THEN or IF/GOSUB is used to conditionally branch to another statement or conditionally call a subroutine based on a comparison of two expressions. Legal relations are:

< less than
> greater than
= equal to
<> not equal to
<= =< less than or equal to
>= => greater than or equal to

If the statement is an IF/GOSUB the subroutine specified will be called if the relation is true and will return to the statement following. Because of A/BASIC's multiple statement line capability, the IF statement can be used as an IF.. THEN.. ELSE function if another statement follows on the same line.

Examples:
```
If N = 100 THEN 1210
If A+B=C*D  GOSUB 5500
IF x<= 200 THEN 240  :  GOTO 1100
IF A$=B$ THEN 300
```

ON ERROR GOTO Statement

SYNTAX: ON ERROR GOTO
ON ERROR GOTO line#

This statement provides a error "trap" - the capability to transfer program control when an error occurs.

When an ON ERROR GOTO statement is executed A/BASIC saves the address of the line number specified in a temporary location. If any detectable error occurs during execution of following statements, the program will transfer to the line number given in the ON ERROR GOTO statement last executed. This would normally be the line number where an error recovery routine begins.

If the ON ERROR GOTO statement is used WITHOUT a line number specified, it has the effect of "turning off" the error trap - errors in following statements will cause the program to stop and an error message to be displayed.

After an error has been detected, the ERR function may be used to access a value which is an error code identifying the type of error which most recently occured. The exact error codes are related to the error codes used by the host operating system and are listed in the appendix.

Example of usage:

```
100 ON ERROR GOTO 500
120 INPUT A(N)
400 N = N+1 : IF N=50 THEN 600 : GOTO 120
500 PRINT "ILLEGAL INPUT ERROR - RETYPE"
600 GOTO 120
```

On-Goto/on-Gosub Statements

Syntax: ON <expr> GOTO <line #>, <line #>, . . . , <line #>
ON <expr> GOSUB <line #>, <line #>, . . . , <line #>

Description: The expression is evaluated and one line number in the list corresponding to the value is selected for a branch or subroutine call, i.e., if the expression evaluates to 5, the fifth line number is used. If the result of the expression is less than specified, the next statement is executed.

Examples: ON A*(B+C) GOTO 200,350,110,250,350
ON N GOSUB 500,510,520,500,100

Stop Statement

Syntax: STOP

Description: The STOP statement is used to terminate execution of a program. A message of the form

STOP <line #>

is displayed and control is returned to the interpreter command mode.

INPUT/OUTPUT STATEMENTS

All input and output statements use a buffer for intermediate storage of data. It may contain up to 128 characters.

NOTE: A special form of all input/output statements designed for buffer direct input/output using the special string variable BUF$ is described in the STRING PROCESSING section.

Input Statement

Syntax: INPUT <var>, . . . ,<var>
Description: This statement causes prompt and space on the terminal device, then reads characters into the input buffer until 128 characters have been read or a carriage return symbol is read. A carriage return/line feed is printed when the last character in input.
Entry of a CONTROL X will print *DEL* and CR/LF and reset the buffer. A CONTROL O will backspace in the buffer and echo the delated characters.

The variables specified <var> may be numeric or string, subscripted or simple type. When the program is "looking for" a number from the current position in the input buffer, it will skip leading spaces, if any and read a minus sign (if any) and up to five number characters. The numeric field is terminated by a space, comma, or end of line. If a non-digit character is read, or any other illegal condition a value of zero will be returned for the number.

If a string-type field is being processed, characters from the current position will be accepted including blanks until the variable field is terminated by a comma or end of line, or when it is "full". If no characters are available, a null string will be returned.

Examples:

```
INPUT  A,B,S$,B$
INPUT  A(N+!,M-1),B,A(4,N)
INPUT  A$(N),B$(N+1),D$
INPUT  B
```

Print Statement

Syntax: PRINT <out spec> <delimiter> . . . <delimiter> <out spec>

Description: This statement processes the list of out spec's and puts the appropriate characters in the buffer. The buffer is then output to the terminal device.

An <out spec> may be a string expression or a numeric expression, or the output function TAB expr which inserts spaces in the buffer until the position <expr> is reached. Each item in the list is seperated by a delimiter which is a comma or semicolon. The buffer is divided into sixteen 8-character zones, which are effectively tab stops every eighth position. If comma is used as a delimiter, the next item will begin at the first position of the next zone. If a semicolon is used, NO spacing will occur. A semicolon at the end of a Print statement will inhibit printing of a carriage/return/line feed at the end of the line.

Examples:

```
PRINT A,B;C
PRINT A$(N);A$(N+1)
PRINT A,A$,B,B$
PRINT TAB(N=1),Z4
PRINT A;B;C;
PRINT A$;TAB(N+M);B$
```

A/BASIC DISK INPUT/OUTPUT OPERATIONS

A/BASIC uses the facilities of the host operating system for input and output to sequential disk files. In order for programs generated by A/BASIC to operate properly when disk functions are used, the memory resident facilites of the DOS must be present.

Disk I/O in A/BASIC is channel-oriented meaning a file to be used for input or output is "opened" and assigned a channel number by which all further operations are performed. A/BASIC supports up to 10 channels which is the maximum number of files that may be open at any time.

All disk file names are defined identically to the operating system's file name conventions, i.e. the file name specifications are identical. With the exception of the CHAIN statement, all files used by A/BASIC are ASCII text files.

Many of the A/BASIC disk operations use the same DOS subroutines as would be used by assembly-language programs so information as to disk operations listed in the disk system manufacturer's software manual will generally apply.

Note: In the descriptions of disk statements that follow the term "filnum" refers to a channel number which is a constant which may range from Ø to 9.

OPEN Statement

SYNTAX: OPEN #filnum,strexp

This statement is used to open (initialize) a file for input or output, and assign the file a channel number. One and only one file may be open on a particular channel at a time. The DOS requires that a particular file be open for either read or write, not both. The OPEN statement will first search the file directory on the specified drive and open the file for read if a file by the name specified is found. If no such file is found, a file with the name specified will be opened (created) for write operation. The STATUS function may be used after a file is opened to determine its read or write status is necessary.

If an error occured during the open operation, the line specified by the last ON ERROR GOTO statement to be executed will be transferred to. The ERR function may be used to determine the error type.

An OPEN statement must be used prior to any I/O to the particular channel. Also note that it is the only time that the file name is explicity used. All further references to the particular file as long as it is open is by means of the channel number.

The channel number must be a constant integer from Ø to 9. The file name specification is a string constant, variable or expression and must conform to the DOS requirements for legal file name specifications (drive, name, extension, etc.)

Below are some examples of legal usage of the OPEN statement:

```
OPEN #Ø,"TEST"          Opens the file TEST and assigns to channel Ø

A$="MASTER"
OPEN #2,A$              Opens the file MASTER on channel w

A$="MASTER"
OPEN#2,A$+"TXT"         Opens the file MASTER.TXT on drive #1 and
                        assigns it channel 2.
```

Close Statement

SYNTAX: CLOSE #filnum

This statement is used to close a file after I/O operations are complete and releases the channel. The file on the channel is closed for either read or write automatically. The channel is then available for reassignment to another file if desired.

Examples of legal usage:

```
CLOSE #2
```

READ Statement

SYNTAX: READ #filnum,varlist

The read statement causes the next record of the read file on the channel specified to be read into the A/BASIC I/O buffer. After the record is read, data items corresponding to the items in the variable list will be taken in order and stored in the appropriate variable locations. The variable list may include numeric or string type which may use subscripted variables.

The number of items given in the variable list should agree with the number of items on the disk data record (except as noted below).
The file must be open for read and must be an ASCII text-type file. Disk records are variable length.

Rules for reading different data types: the following rules apply to and define the result of reading items from a disk record under various circumstances.

NEXT VARIABLE IN	NEXT ITEM IN BUFFER (RECORD) IS ..		
LIST IS ...	NUMERIC	STRING	EMPTY
NUMERIC TYPE	NUMBER	NUMBER if string contains legal decimal chars.	ERROR
STRING TYPE	STRING OF DIGITS	STRING	NULL STRING

If there are more items on the record than variables in the list, they will be ignored. Note that numeric variables are not stored as binary bytes, rather they are converted to ASCII character representation before being written and converted to binary after being read.

WRITE Statement

Syntax: WRITE #filnum,outlist

The WRITE statement causes a record containing the data specified to be written on the disk file as a single record. The file must be open for use as a write file (see OPEN).

The output list is a sequence of string or numeric constants, variables or expressions separated by commas. Each element in the list is considered to be an "item" within the disk record to be written. Strings and numeric values are written in ASCII form. Type conversion is automatic for numbers. The resulting record length must be less than or equal to a total of 128 bytes.

Examples of legal usage:

```
WRITE #3,A,B,C
WRITE #6,"DATA",N$,C$,Z$,T
WRITE #6,400+Z,A$+MID$(B$,4,M),M/2
WRITE #2,"MASTER FILE NUMBER "+STR$(N)
```

Results of writing disk record:

Each item is written on the record with a comma item seperator. The records are variable length and use a carriage return character as an end of record terminator. The statement (assume variable N=10):

```
WRITE #2,25,-ØØ,"WORD",N
```

produces a disk record which has the following format (in hex):

32	35	2C	2D	34	3Ø	3Ø	2C	57	4F	52	44	2C	31	3Ø	0D
2	5	,	-	4	0	0	,	W	0	R	D	,	1	Ø	EOR

CHAIN Statement

Syntax: CHAIN strexp

This statement allows the BASIC program to load and begin execution of another (or an additional) program. The program name is specified by the string expression.

The program to be CHAINed must have the following characteristics:

1) It MUST have been saved on the disk as a binary format program. A/BASIC produces HEX format object programs so A/BASIC-produced programs must be loaded and then saved in binary before being CHAINed.
2) A transfer address (starting address) must have been specified when the binary program was saved.

WARNING: THIS STATEMENT LOADS A NEW MACHINE LANGUAGE PROGRAM AND CAN POSSIBLY WRITE OVER THE INTERPRETER!!

RESTORE Statement

SCRATCH Statement

SYNTAX: RESTORE #filnum, ... ,#filnum
SCRATCH #filnum, ... ,#filnum

These statements are used to close and then reopen files on the specified channel(s). The channels must have been previously OPENed.

The RESTORE statement closes a file open for either read or write and opens it again for read. This is analogous to a "rewind" operation.

The SCRATCH statement closes a file open for either read or write and opens it for write.
WARNING: The SCRATCH statement destroys the file being SCRATCHed!!! It in effect deletes (destroys) the file and opens a new file with the same name. When a file is SCRATCHed all previous data is lost.

Because of the sequential nature of files supported by the DOS, these statements are used to "reposition" the internal "pointer" to the next record to read from or written to.

Examples of legal usage:

```
SCRATCH #2,#8
RESTORE #3
```

KILL Statement

SYNTAX: KILL<stringexpr>

The file name specified is PERMANENTLY DELETED from the system. Use with great care as the file may not be recovered.
Example:

```
KILL "TEMP3"
KILL A$
```

DISK FUNCTIONS

The following functions are available for use with disk input/output operations. All operate as the other BASIC functions and return a numeric-type result.

EOF Function Syntax: EOF(#filnum)

Returns a value of 1 if an end of file condition exists on the file assigned to the channel specified, otherwise returns Ø.
Example: IF EOF(#4) = 1 THEN 55Ø

FREE Function Syntax: FREE(drive number)

Returns the number of sectors that are available (not used by existing files) on the disk drive unit specified.

FILSIZ Function Syntax: FILSIZ(#filnum)

This function returns the current length of the file specified in sectors.
Example: IF FILSIZ(#4) = 35 THEN 44Ø

STATUS Function Syntax: STATUS(#filnum)

Returns the current status of the file specified as follows:

Ø = File is not currently open
1 = File is open for read
2 = File is open for write

Examples:

```
ON STATUS(#8) GOTO 47Ø,12Ø
IF STATUS(#1) = 1 THEN 61Ø
N2=STATUS(#5)
```

DUMMY STATEMENTS

Some compiler statement types cannot be executed by the interpreter, however they will be accepted and treated as "comments" (similar to REM). A warning message will be printed when these statements are encountered. This feature is included so the interpreter can be used to prepare an entire compiler source file.

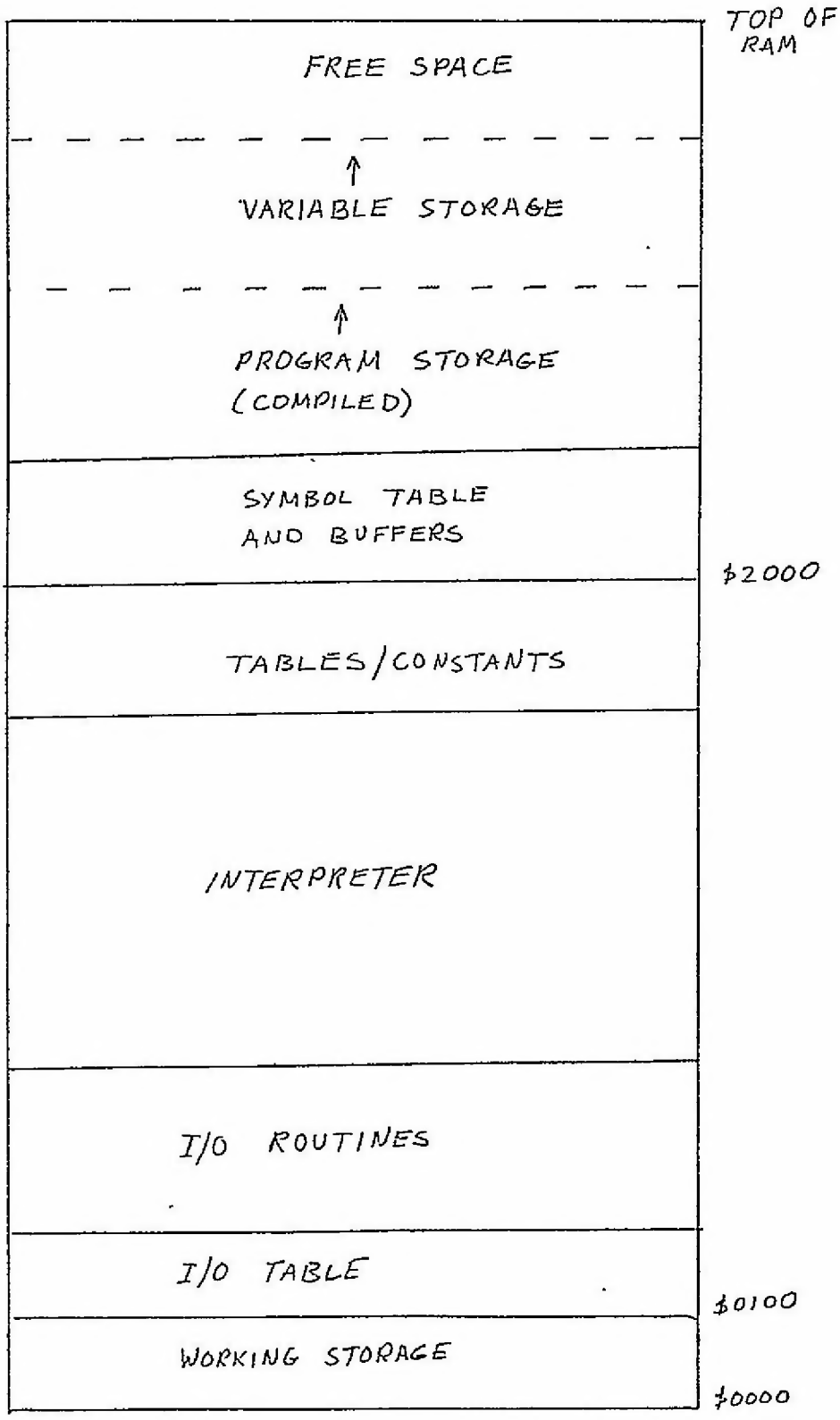
Dummy Statements:

ON NMI GOTO	GEN
ON IRQ GOTO	IRQ ON
ORG	IRQ OFF
OPT	RETI
STACK	TASK
SWITCH	

A/BASIC INTERPRETER ERROR CODES

ERROR NUMBER	MEANING
04	Syntax error
05	Var name missing
06	Equal Sign missing
07	Undefined line reference
08	Parenthesis missing
09	Numeric expression missing or in error
10	Destination line number missing or in error
11	Number missing
12	Misnested FOR/NEXT loop
13	Symbol table overflow
15	Missing or illegal use of realtional operator
17	Quote missing at end of string
18	Illegal type or missing counter variable in FOR state
19	Redefined array
20	Error in array specification; subscript missing, or too many subscripts
21	Error in array specification; subscript 0 or larger than 255
23	Reference to undeclared array
24	Subscript error
29	String expression missing or in error
30	Too many FOR/NEXT loops active - max is 20
34	GOTO or GOSUB missing
36	Error in DISK I/O LIST
37	Return without Gosub
62	Multiply overflow
63	Divide by zero
64	Conversion error

MEMORY MAP


TOP OF RAM
FREE SPACE
VARIABLE STORAGE
PROGRAM STORAGE (COMPILED)
SYMBOL TABLE AND BUFFERS
$2000
TABLES/CONSTANTS
INTERPRETER
I/O ROUTINES
I/O TABLE
$0100
WORKING STORAGE
$0000