A/BASIC

COMPILER REFERENCE MANUAL

Table of Contents

Table of Contents

A/BASIC VOCABULARY LIST

The terms defined below may be unfamiliar to some programmers or used in a special context in this manual.

ALLOCATION: The process of assignment of a specific memory address to variables or machine instructions.

CODE GENERATION: The process of creating machine language instructions.

COMPILE-TIME: Used to describe the time during which the BASIC program is being processed by the compiler.

LIBRARY: A collection of subroutines within the compiler (BASLIB) which are used to generate subroutines within the machine language program. It includes "images" of subroutines for mathematical functions, string processing, input/output, array operations, etc.

LINE REFERENCE: A reference from a statement to a line number which is that of another line.

LINE REFERENCE TABLE: A table which is kept by the compiler which is used to store the correspondences between referenced line and the memory address assigned to the line.

LINKER/EDITOR: A portion of the A/BASIC compiler program which places subroutines which are required in the object program. The images are obtained from the library and the linker/editor inserts absoluted address of the subroutines in the program. Only one copy of a subroutine will ever be generated in a program, and only those that are required by that specific program.

OBJECT CODE: The machine-language instructions generated by the compiler.

OBJECT FILE: The tape or other media file that the compiler has written the machine language program on.

OBJECT PROGRAM: The machine language program produced by the compiler from the BASIC source program.

PASS: A complete scan of the source program. A/BASIC is a two pass compiler so it must completely read the source program twice.

REAL-TIME EXECUTIVE: The portion of the OS-9 Operating System that schedules and controls task execution.

RUN-TIME: Used to describe events or the time during which the machine language program is being executed.

SOURCE FILE: The tape or other media containing the A/BASIC program which is to be compiled.

SOURCE PROGRAM: The BASIC program text to be compiled.

SYMBOL TABLE: A table kept by the compiler during compilation that contains information about the correspondence between variable names and assigned memory addresses, and type of variable.

SYNTAX: Rules for proper construction of parts of BASIC statements.

TWO'S COMPLIMENT: A method of binary representation of numbers where negative numbers are represented as the result of subtracting the absolute value of the number from zero.

Introduction

A/BASIC is an optimizing two-pass BASIC compiler for the M6809 family of microprocessors which converts programs written in BASIC to 6809 native code. It is not meant to teach the BASIC language to a beginner, but as a powerful tool for someone experienced in the use of BASIC.

A/BASIC is oriented towards applications previously programmed in assembly language due to its capability to produce extremely fast, compact object programs.

The compiler's output can be run as a stand-alone RAM, ROM or PROM based program which may be run without any run-time package. A built-in linker/editor automatically selects subroutines from A/BASIC's internal library and inserts one and only one copy of subroutines required directly into the object program.

Depending on the specific program, A/BASIC can produce programs which may reflect a 50 to 1000 times speed improvement over interpreters with a significant memory savings - typically 25-50% less memory required.

A/BASIC also contains statements for creating, manipulating and performing I/O to disk data files which make it suitable for a wide range of system programming applications.

GENERAL INFORMATION ABOUT A/BASIC

A/BASIC is a two-pass true compiler that directly converts BASIC language statements to 6809 machine language instructions. Compared to standard BASIC interpreters, A/BASIC programs run from 50 to 1000 times faster and usually require substantially less memory to run. The compiler output is "ROMable" meaning it can be placed into ROMS directly.

A/BASIC is used on BASIC program files which you wish to compile into object code. These files must be in keeping with proper A/BASIC syntax. You would first use an Editor (such as FHL's ED) to create these files.

Then the compiler reads the BASIC program from disk and outputs the machine language program on another disk file. The compiler must scan the source program twice to operate correctly. During each pass, the source program is read line by line and corresponding machine language generated.

The first pass is almost identical to the second in that the compiler performs almost the same functions. However, during the first pass the compiler does not "know" all of the eventual memory address of statements and subroutines that have not yet been processed. Therefore the primary purpose of the first pass is to build two major tables, the symbol table and line reference table, that will be used to locate variable, statement and subroutine addresses during

Introduction

A/BASIC is an optimizing two-pass BASIC compiler for the M6809 family of microprocessors which converts programs written in BASIC to 6809 native code. It is not meant to teach the BASIC language to a beginner, but as a powerful tool for someone experienced in the use of BASIC.

A/BASIC is oriented towards applications previously programmed in assembly language due to its capability to produce extremely fast, compact object programs.

The compiler's output can be run as a stand-alone RAM, ROM or PROM based program which may be run without any run-time package. A built-in linker/editor automatically selects subroutines from A/BASIC's internal library and inserts one and only one copy of subroutines required directly into the object program.

Depending on the specific program, A/BASIC can produce programs which may reflect a 50 to 1000 times speed improvement over interpreters with a significant memory savings - typically 25-50% less memory required.

A/BASIC also contains statements for creating, manipulating and performing I/O to disk data files which make it suitable for a wide range of system programming applications.

GENERAL INFORMATION ABOUT A/BASIC

A/BASIC is a two-pass true compiler that directly converts BASIC language statements to 6809 machine language instructions. Compared to standard BASIC interpreters, A/BASIC programs run from 50 to 1000 times faster and usually require substantially less memory to run. The compiler output is "ROMable" meaning it can be placed into ROMS directly.

A/BASIC is used on BASIC program files which you wish to compile into object code. These files must be in keeping with proper A/BASIC syntax. You would first use an Editor (such as FHL's ED) to create these files.

Then the compiler reads the BASIC program from disk and outputs the machine language program on another disk file. The compiler must scan the source program twice to operate correctly. During each pass, the source program is read line by line and corresponding machine language generated.

The first pass is almost identical to the second in that the compiler performs almost the same functions. However, during the first pass the compiler does not "know" all of the eventual memory address of statements and subroutines that have not yet been processed. Therefore the primary purpose of the first pass is to build two major tables, the symbol table and line reference table, that will be used to locate variable, statement and subroutine addresses during

the second pass.

During the second pass, each source statement is read and machine language and listing lines are output. Each source line is compiled individually with the help of information stored in the symbol and line reference tables. The three major parts of the compiler are the **statement processor** which decodes the source statements and breaks them down to their individual syntactical elements, the two **expression processors** (string and numeric) which convert expressions to an internal representation and optimize the order of operands and operators, and the **code generator** which converts the expressions from the internal format to 6809 machine language and also performs another level of optimization by selecting optimum addressing modes and instruction sequences (such as CLRA instead of LDA #0).

During compilation of the main program the compiler notes which special subroutines from the built-in library (BASLIB) will be required for the specific program. The BASLIB library is a collection of special subroutines that perform functions that are not available in the 6809 instruction set (such as divide) or are commonly referred to throughout a program (such as input/output and string operations). The BASLIB subroutines are short, fast and position-independent (the total less than 1k). At the end of the pass, the compiler automatically appends the required BASLIB subroutines to the program and another part of the compiler (the linker) inserts the necessary subroutine calls within the program. One copy only of those subroutines which are required for the particular program are included. This eliminates the need for the typical "run-time packages" used by other compilers and adds to the ROMability of A/BASIC output code.

The A/BASIC Library Runtime Source is available for those who need it. The cost is $75.00 and may be obtained from:

Fair and Associates
1952 So. Holland St.
Lakewood, CO 80227
(303) 985-7419

Consultation on A/BASIC programs is also available on a fee basis from the above address. Please do not contact FHL in regards to either of the above as we will only refer you to Fair and Associates.

HINTS FOR PRODUCING GOOD A/BASIC PROGRAMS

1. **SUBROUTINE** Use subroutines wherever possible, even for one line which is used repeatedly. This will result in significantly shorter machine code.

2. **USE THE DIRECT PAGE** Use the Base statement (and DIM if necessary) to place your commonly used simple variables in the 6809 direct page ($0030 to $00FF). Put big arrays and string variables elsewhere. This will permit the compiler to use the direct addressing mode which produces shorter, faster instructions.

3. **USE REMARKS GENEROUSLY** With A/BASIC, REM statements do not affect the machine language program size or speed at all. Use comments to improve the internal documentation and readability of your programs. It pays in the long run to produce well documented programs that can be referred to or modified by yourself or others.

4. **READ THIS MANUAL CAREFULLY** There are many treats buried in this document (and some potential pitfalls as well!) If A/BASIC is your first experience with a compiler read this book again! 90% of the questions people write or call us about are answered in this manual.

5. **USE THE LISTING TO HELP DEBUG PROGRAMS** If your A/BASIC program compiled without errors but does not perform as expected, you almost certainly have an algorithmic fault (otherwise known as programmer error). The A/BASIC listing provides you with much valuable information that can be used to find run-time errors in conjunction with your interactive debugger program. The statement addresses on the listing can be used to place breakpoints, and the symbol table dump (OPT S) shows variable addresses that can be examined with the monitor's memory change function.

6. **GET TO KNOW YOUR COMPUTER** If you have not had experience with the 6809's machine language, take the time to acquire some understanding of it. A/BASIC offers a convenient way: Write some very short programs, compile them and observe how A/BASIC generates corresponding machine language by using OPT H

O.K. HERE'S MY DISK - WHAT DO I DO WITH IT?

The first thing you should do with your A/BASIC disk is to make a copy of it onto a work disk. This should be done on a newly formatted 35 or 40 track disk (depending on your drive capacity) which can then be used for storage and manipulation of A/BASIC files.

HERE'S HOW (on a FLEX system)

With a Single Drive:

1. Boot up your Flex operating system
2. Format a new disk, using the Flex command:NEWDISK

3. Reinsert your Flex disk and use the Flex command, SDC to copy A/BASIC onto the newly formatted disk. (SDC ABASIC.CMD)

4. You may then either copy your Editor onto the newly made work disk, copy the Flex BUILD command onto the disk, or copy each individual program file to be compiled onto the work dısk using the Flex SDC command.

With Multiple Drives:

1. Boot up Flex in Drive # 0
2. Format a new disk in Drive # 1 (NEWDISK 1)
3. Use Flex command COPY to make a copy of your A/BASIC disk.

HERE'S HOW (on an OS-9 system)

With A Single Drive

1. Boot up your system
2. Format a new disk using FORMAT.
3. Load 'LOAD' (Load the Load program into memory).
4. Insert the A/BASIC disk.
5. Load 'A/BASIC' (Load the compiler into memory).
6. Insert your new dıskette (or Destination disk).
7. Save 'ABASIC' (Save the compiler on the Destination disk).

With Multiple Drives

1. Boot up your system.
2. Format a new disk.
3. Copy the files from the master to the new disk.

A/BASIC PROGRAM STRUCTURE (In other words, your programs to be compiled must follow this syntax!)

The first step in producing an A/BASIC program is to use the system's text editor to create a source file which will contain your A/BASIC program. A/BASIC accepts the file formats used by most editors.

An A/BASIC program consists of a series of source lines. A source line may optionally 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 sepparate the statements. A source line may contain up to 80 characters.

Line numbers are positive decimal numbers up to four digits. 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). Unlike interpreters, REMark statements do not affect program size and may be used generously.

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

Example of program structure:

We'll start with a tough one. It's an averaging program with a file name of, what else but AVG.TXT

```
100 PRINT "THIS PROGRAMS AVERAGES A SERIES OF NUMBERS"
    PRINT "HOW MANY NUMBERS ":INPUT N
    PRINT "WHAT IS THE FIRST NUMBER? ":INPUT T
    FOR X=2 TO N:PRINT "NEXT NUMBER?": INPUT I
    T=T+I :NEXT X
    PRINT:PRINT "THE AVERAGE IS";T/N
    PRINT "DO YOU WANT TO CONTINUE?": INPUT A$
    IF A$="YES" THEN 100
    STOP
    END
```

* Did you type this valuable program in? It's sure to bring you hours of enjoyment!!

COMPILATION PROCEDURES

Now hopefully you've got a work disk with ABASIC on it, and a fun little averaging program in a file called AVG.TXT on there too. Oh boy.

The compiler is then called by a system command such as

ABASIC,SOURCE,OBJECT,+OPTIONS (for FLEX)

ABASIC SOURCE O=OBJECT OPTIONS (for OS-9)

The above format specifies the file "SOURCE" as being the "uncompiled" file. This must be a Text file (i.e. contain an A/BASIC source program). This will default to a .TXT extension under FLEX.

The file called "OBJECT" is the file into which the "compiled" version of the source file is put. The object file will default to a .BIN (Binary Machine Code) extension under FLEX or the current Execution Directory under OS-9.

NOTE: Specifying the Object file to have the same name as an already existing file will cause the existing file to be deleted.

"OPTIONS" is an optional list of compile-time options:

H - Generates Hex Code in Listing
L - No Listing
S - Generate Symbol Table

So in our example:

ABASIC,AVG,AVG,+L or
ABASIC AVG.TXT O=AVG.BIN L

will take the sample source program stored in AVG.TXT and compile it and store it in another file called AVG.BIN, and generate no listing. Try it-- Go back to the operating system and type AVG.BIN

What actually happens is that A/BASIC is loaded by the DOS and begins compilation. During the second pass, a formatted listing is produced which will include any error messages detected during this pass (See COMPILE-TIME ERRORS). The listing usually consists of three fields: The first is the Hex address where the object code starts for the BASIC source line; the second field is the statement line number if any; and the last field is the BASIC source line.

After compilation is complete, the program statistics, load map and symbol table (if the program was compiled with the S option) are printed. The load map lists the names and addresses (main entry point) of the subroutine packages the compiler selected and included in a particular program.

ERROR MESSAGES AND PROCESSING

When A/BASIC detects an error in the source program during either the first or second pass it will print the source line in error and a message with an error code (the codes are listed in the appendix). The line below the erroneous source line will have an up arrow showing the approximate position of the error. This error location is about 95% accurate.

When an error is detected on a source line the compiler will not process the line further even if it is a multiple statement line, so the rest of the source line should be examined carefully for possible undetected errors.

A/BASIC COMPILE-TIME ERROR CODES

02 Line number duplicated or out of sequence
03 Unrecognized statement
04 Syntax error
05 Variable name missing or in error
06 Equal sign missing
07 Undefined line reference: GOTO or GOSUB to nonexistant line number
08 Right parentheses missing or misnested
09 Operand missing in expression
10 Destination line number missing or in error
11 Number missing
12 Misnested FOR/NEXT loop(s)
13 Symbol table overflow *
14 Illegal task number - must be 0 to 15
15 Missing or illegal usage of relational operator(s)
16 Delimiter (, or ;) missing
17 Quote missing at end of string
18 Illegal type or missing counter variable in FOR statement
19 Redefined array
20 Error in array specification: subscript missing; too many subscripts
21 Error in array specification: subscript zero or larger than 255
22 Variable storage overflow, tried to allocate past $FFFF
23 Reference to undeclared array
24 Subscript error
25 Missing, illegal type or incorrect function argument (numeric)
26 Illegal option
27 Unrecognized operator in string expression
28 Concatenation operator (+) missing
29 Missing, illegal type or incorrect function argument (string)
30 Too many FOR/NEXT loops active - max is 16 Line reference table overflow *
32 Program storage overflow - tried to allocate past $FFFF
34 GOTO or GOSUB missing
35 Illegal channel number: must be 0 to 9
36 Error in disk I/O list

* These error types are not program errors. If more system memory is available the tables may be expanded to include more entries.

A/BASIC RUN-TIME ERROR CODES

The ERR function will return one of the following codes after an error occurs at run-time:

ERRORS 0 TO 31	DISK ERRORS The codes used are identical to those used by the host DOS and may be found in the system's DOS manual. All codes may not be implemented by DOS.
ERROR 32	MULTIPLY OVERFLOW The result of a multiplication exceeds the range +32767 to -32768. The result was the low order 16 bits of the result and the high order 16 bytes are saved in the fast scratch area.
ERROR 33	DIVIDE ERROR A divide with a zero divisor was attempted. A result of zero was returned.
ERROR 34	CONVERSION ERROR The BASLIB ASCII-to-binary conversion read illegal, oversize or no input. A value of zero was returned.

NUMBERS

A/BASIC's numeric data type is internally represented as 16 bit (2 byte) two's compliment integers. This permits an equivalent decimal range of +32767 to -32768. This data representation is quite natural to the 6809's machine instruction set which allows A/BASIC to produce extremely fast and compact machine code.

Because the compiler supports boolean (True/False) operations, unsigned 16 bit binary numbers may also be used for many functions. The range for these are: 0 to +65535. These numbers are used for referencing memory address 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 or a pound sign # to represent the logical compliment (1's compliment or boolean NOT).

Examples of legal number constants:

```
200  -5000  $0100  -3000  12345  #1  #$5000  $FFF0  #$C0F1
```

Examples of ILLEGAL NUMBERS:

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

Because binary numbers are represented in either unsigned or 2's compliment form, 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  $FFFF  #0  65535
```

NUMERIC VARIABLES

Legal numeric variable names in A/BASIC consist of a single letter A-Z or a single letter and a digit 0-9. The following are legal variable names:

X N R2 Z9 A0 P1

If declared in a DIM statement, numeric variables may be arrays of one or two dimensions. The maximum subscript sizes is 255, therefore the largest one-dimensional array has 255 elements and the largest two-dimensional array has 255*255=65025 elements. Subscripts begin at 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. A/BASIC does not perform run-time subscript checking for overrange errors (which would cost considerably in terms of program size and speed).

References to two-dimensional arrays require the program to perform a multiplication to calculate the actual element address. Even though A/BASIC uses the internal fast 8 by 8 bit multiply for array address calculations it takes about 32 MPU cycles minimum to access a two-dimensional element as opposed to about 19 MPU cycles for a one-dimensional access.

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-dimentional 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 arithmetic expressions. The boolean operators perform a bit-by-bit operation across all 16 bits of the operands.

The order of operations determine in which order A/BASIC processes the parts of expressions. The compiler converts arithmetic expressions to an internal form during compilation and rearranges the expression by operator precedence to produce machine instructions which are as short and fast as possible. Operators 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("454")
```

ARITHMETIC FUNCTIONS

A/BASIC supports the following numeric 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 0 to +32767. If an argument is supplied, it is evaluated and used to "seed" the random number generator (randomize it).

PEEK(expr)

The expression is evaluated and used as an address. The single byte at the address is returned as the low-order byte of the result. The high order byte of the result is always zero, so the value returned will be in the range 0 to 255.

POS

The current character position in the output buffer (print position).

SWAP(expr)

Byte swap of the result of the argument. (The high order byte is exchanged with the low order byte.)

ERR

Returns error code of the most recent error condition. See "RUN TIME ERROR CODES" in the Appendix for specifics.

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. A/BASIC will detect this error (see ON ERROR GOTO) and preserve the high-order 16 bits of the correct 2's compliment result at address $002B and $002C.

Division attempted with a divisor of zero will also produce an error which is detected at run-time with the ON ERROR GOTO statement.

MULTIPLE PRECISION ARITHMETIC

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 convention 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
    A1=A1+1 :Rem add 1 to ms bytes for carry
200 A1=A1+B1 :Rem add ms bytes
    RETURN
```

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

```
100 A2=A2-B2  : ON OVR 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 use, 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 the 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 as it is being compiled without having been previously declared in a DIM statement the compiler 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:

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

STRING CONCATENATION

The string concatenation operator + is used to join strings to form a new string or string expression. 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=0 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 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 that evaluation of function arguments are performed before concatenation.

At run-time, string operations are typically performed on strings moved to a string buffer. This buffer is normally allocated by the compiler to be 255 bytes long. Because this is always the last data storage allocated by the compiler, any memory available beyond this area may be used to allow automatic buffer expansion if operations on extremely complex string expressions are involved, or if string variables or constants have a long length.

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$,Z4),X+2)+C$
```

STRING COMPARISON

Strings may be compared in an IF expression the same as their numeric counterparts. Each character in the string is numerically evaluated as an ASCII character.

Example: IF A$ <= B$ THEN C$=B$

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

X$ 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$. An example of this functions is: RIGHT$("EXAMPLE",3) which produces "PLE".

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

TRM$(X$) is a function which emoves 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 STR$(N) function will result in a run-time error detectable by the ON ERROR GOTO function if its argument is not legal or convertable to a string.

The following functions have string argument(s) and produce a result which is 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 0.

SUBSTR(X$,Y$) is a substring search function which searches for the string X$ in the string Y$. If and identical substring is found the function will return a number which is the postion of the first character of the substring in the target string. If the substring is not found the function returns a value of 0. For example, the function SUBSTR("EXAMPLE","PL") returns a value of 5. SUBSTR("EXAMPLE","NOT") returns a value of 0.

VAL(X$) converts a string of characters in the form of decimal digits and (optionally a leading minus sign) to a numeric value. This has the inverse effect of STR$. If the string argument is not a legal conversion string (it has too many, no-decimal or no digit characters) a runtime 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 format conventions cause commas which are part of the input data to break up what may actually be one long string, etc. A/BASIC has a special string variable, BUF$ which is defined to be the entire contents of the run-time I/O buffer. BUF$ may be used as any other string variable, and may be up to 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)
```

COMPILER DIRECTIVE STATEMENTS

ORG AND BASE STATEMENTS

SYNTAX:

```
ORG=<addr>
BASE=<addr>
```

These statement types are used to control how A/BASIC assigns memory in the object program. The ORG statement is used to assign starting addresses for the object code and the BASE statement is used to define the addresses used for variable storage.

Both statement types may be used as often as desired so memory assignments for program and data storage may be segmented as desired.

A/BASIC uses two internal "pointers" that control how run-time memory is allocated. The "object code pointer" always maintains the address where the next instructions generated by the compiler will be stored. The ORG statement assigns a value to this pointer. When A/BASIC is first entered, a default value of $1000 (for FLEX) and $0000 (for OS-9) is assigned to the pointer so unless an ORG statement is processed before the first executable BASIC statement, this will be the program's default starting address.

For example, the statement:

```
ORG=$2400
```

will cause instructions generated for following BASIC statements to begin their address at $2400. The ORG statement may be used to create "modules" at different address within a single program.

The BASE statement is also used to control memory assignment in a similar manner but it applies to allocation of RAM for variable storage. An internal "data address pointer" is maintained by A/BASIC to hold the next address available (at run-time) for variable or temporary storage. It is initialized by default to address $0030.

A/BASIC assigns RAM corresponding to BASIC variables the first time they are encountered in the source program at compilation time. When a "new" variable name is encountered, A/BASIC assigns the variable run-time storage corresponding to the current value of the data address pointer which is then updated by increasing it by the size of the variable storage assigned. It therefore is again pointing to the next available RAM location.

CONT.

An important function of the BASE statement is to allow specific memory assignments for specific variable names. Reasons for this application are:

1) To take advantage of the 6809's "direct page" addressing mode. When commonly used variables are located in the page of memory from $0000 to $00FF the compiler uses the direct addressing mode which can reduce program size and increase execution speed substantially.

2) To assign RAM consistant with actual RAM addresses that are available in the computer the software is to run on.

3) To assign specific variable names and types with memory addresses which have special functions. For example address of ACIAs, PIAs or other interface registers may be given BASIC variable names. A common type of "trick" is to declare the memory used by a video display (memory-mapped) as a BASIC string array which permits fast, simple updates to the video image.

Sometimes it is useful to declare a variable without generating code at the time it is declared. If the variable is an array, the DIM statement may be used. If it is a simple type, the DIM statement declaration with a size of one may be used for a declaration. For example, to assign the address $8010 to the variable P1 the following sequence may be used (assuming P1 had not been referenced previously in the program):

```
BASE=$8010
DIM P1(1)
```

WARNING: Because of the Memory Management characteristics of OS-9 Level II the user of that system should be careful of using direct memory mapping in that environment.

Note: The working data area of the compilier has been relocated to $2C00 and the code has been moved to $0100 as per FLEX convention. This will also aid in the extending of the tables for those who need it.

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 arrays 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 string size. 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 <u>not</u> an array. B$ would be used in the program with <u>one</u> subscript because it is a one-dimensional array. For example:

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

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 particular point, but without generating code. 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 by internally assuming a subscript of (1) for a one dimensional array or (1,1) for a two-dimensional array. 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 address $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:

```
S=Q
```

or to read the ACIA:

```
N=A
```

WARNING: Because of the Memory Management characteristics of OS-9 Level II, the user of that system should be careful of using direct memory mapping in that environment.

END STATEMENT

Description: The END statement is the last statement of the source program. It causes compilation to cease and any statements following are ignored. If the BASIC source program omits an END statement and end-of-file is read on the source file, END will be automatically assumed.

End does not result in generation of code. If it is desired to return to the host operating system at run-time, a STOP statement must be used before the END.

NAME STATEMENT (For OS-9 only)

Description: The NAME statement is used to generate the module name for the executable code being generated. This name should not be confused with the name of the source file or of the object file, it is the module name when the program is loaded into memory. The name statement uses no quotes and must be a string literal. If more than one NAME statement is used in a program, the last one processed will be used. If no NAME statement is found then an error message at the end of compilation will be printed.

Example:

```
NAME TEST.PROGRAM
NAME DOIT
```

REMARK STATEMENT

Description: 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.

PAGE STATEMENT

Description: Causes the compiler listing to skip to the beginning of a new page.

ASSIGNMENT Statements

Arithmetic Assignment

SYNTAX: LET<var>=<expr>
<var>=<expr>

Description: 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(addr)=<expr>

Description: The expression is evaluated, and the result is truncated to a single byte value which is stored at (addr). Addr may be a literal numerical address, a variable, or an expression. If the form POKE(number) is used to specify the address the fastest possible code is generated. The least significant byte of the result is stored.

WARNING...
Because of the memory management characteristics of OS9 Level II the user of that system should be careful of using this statement in that environment.

String Assignment

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 variable assigned, the first N characters only are stored where N is the length of the assigned 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 $E0CC      Call subroutine at address $E0CC
CALL 1024        Call subroutine at decimal addr.
1024
```

WARNING...

Because of the memory management characteristics of OS9 Level II the user of that system should be careful of using this statement in that environment.

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 BASICs due to a looping method that results in extremely fast execution and minimum length. Note the following characteristics of the FOR/NEXT operation:

1. <var> must be a no-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. At compile time, up to 16 loops may be active, and all must be properly nested.

6. The initial and terminating values may be positive or negative. The step value must be positive. To step through a loop in a negative direction, see Example #2 below.

7. The loop will terminate when the counter variable is greater than the terminating value.

Example #1:

```
FOR N=J+1 TO Z/4 STEP X*2
```

Example #2:

To obtain a decrementing value without using a negative step

```
FOR I=1 TO N
PRINT N-I+1
NEXT I
```

GOSUB/RETURN Statement

Syntax:

```
GOSUB <line #>
RETURN
```

Description: The GOSUB statement calls a subroutine starting at the line number specified. If no such line exists, an error message will be generated on the second pass. The machine stack is used for return address linkage. The RETURN statement terminates the subroutine and returns to the line following the calling GOSUB. Subroutines may have multiple entry and return points. The GOSUB and RETURN statements compile directly to BSR and RTS machine instructions, respectively.

IF/THEN Statement

Syntax:

```
IF <expr> <relation> <expr> THEN <line #>
IF <expr> <relation> <expr> GOSUB <line #>
IF <expr> <relation> <expr> THEN <statement>
IF <expr> <relation> <expr> THEN <statement> ELSE
<statement>
```

Description: The IF/THEN, IF/GOSUB or IF/THEN/ELSE is used to conditionally branch to another statement, conditionally call subroutine or conditionally execute a statement 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 R=CHR$(A$)
IF A$<B$ THEN C$=A$ ELSE C$=B$
```

If the relation is true, then the rest of the line is executed.

ON ERROR GOTO Statement

SYNTAX:

```
ON ERROR GOTO
ON ERROR GOTO<LINE#>
```

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

When an ON ERROR GOTO statement is executed the compiler 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 be ignored.

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.

The types of errors that can be detected by ON ERROR GOTO and the types of statements they occur in are listed below:

```
DIVIDE BY ZERO      ARITHMETIC EXPRESSIONS
ASCII-TO-BINARY CONVERSION  INPUT, VAL(X$)
MULTIPLY OVERFLOW   ARITHMETIC EXPRESSIONS
```

Example of usage:

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

When an error occurs, the program will jump out of the offending routine to the error routine (ON ERROR GOTO destination). However, the routine exited would likely be a (nested) subroutine so the MPU stack pointer may be misaligned.

If "local" error routines are necessary within subroutines GEN statements can be used to insert machine instructions to save and restore the stack pointer as needed.

ON ERROR GOTO - Cont'd

Example of an error handling program:

```
200         ON ERROR GOTO 500
            INPUT A,B
300         ON ERROR GOTO 600
            C=A*B
            GOTO 200
500         PRINT "INPUT ERROR - RETYPE"
            GOTO 200
600         PRINT "MULTIPLY ERROR"
            GOTO 200
            END
```

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 500,510,520,530,540
```

STOP Statement

Syntax: STOP

Description: The STOP statement is used to terminate execution of a program by causing an exit to the host operating system. The end statement should not be confused with STOP as STOP will not terminate compilation of the program.

CHAIN Statement

Syntax: CHAIN program

Description: This statement allows an A/BASIC program to load and execute another program.

OS-9: The chain command performs an F$CHAN system call to the program or modual. Because of the nature of the OS-9 operating system CHAIN may be used to call any executable program and paramaters may be passed just as with 'SHELL'.

FLEX: The chain command performs a "LOAD and GO" call to FLEX.

SHELL Statement (For OS-9 only)

Syntax: SHELL <strexp>

Description: This command calls OS9s SHELL processor. It is used to initiate a child process via the FORK call to OS9.

Examples:

```
SHELL "DIR"
A$="DIR E"
B$="/P"
SHELL A$+" >"+B$+" &"
```

GEN Statement

SYNTAX: GEN <number>,<number>, ... ,<number>

Description: The GEN statement allows data or machine language instructions to be directly inserted in the program. The list of values supplied are inserted directly into the object program. If a value given in the list is less than 255 one byte will be generated for that value regardless of leading zeros.

Example:

```
GEN $BD,$E141,$CE,1024   (PRODUCES 6 BYTES)
GEN 0040,$00,32767       (PRODUCES 4 BYTES)
```

INPUT/OUTPUT Statements

All input and output statements use a 129-byte buffer for intermediate storage of data. This buffer may contain up to 128 characters. The buffer is automatically allocated by the compiler after allocation of all other memory space (except the string buffer if used). This buffer is only allocated for programs that use input/output.

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

INPUT Statement

Syntax: INPUT <var>, ... ,<var>

Description: This statement causes code to be generated which prints a ? 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 line feed and nulls are output upon receipt of the carriage return character. At run-time, 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 deleted 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+1,M-1),B,A(4,N)
INPUT A$(N),B$(N+1),D$
INPUT B
```

PRINT Statement

Syntax: PRINT <outspec> <delimiter> ... <delimiter> <outspec>

Description: This statement processes the list of <outspec>s and puts the appropriate characters in the buffer. The buffer is then output to the terminal device.

An <outspec> 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 separated by a delimiter which is a comma or a semicolon. The buffer is divided into sixteen 8-character zones, which are effectively tab stops every eight positions. If a 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 spaceing will occur. A semicolon at the end of a PRINT statement will inhibit printing of a carriage return/linefeed 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$;TAB(N+M);B$
```

DISPLAY Statement

Syntax: DISPLAY <outspec> <delimiter> ... <delimiter> <outspec>

Description: This statement works the same as PRINT except that all numeric quantities are sent out as though they were a CHR$. When display is used to position the cursor be sure to seperate the command values by semi-colons and not commas. The commas would send unwanted spaces to the terminal.

Examples:

```
DISPLAY 10;13  Print a carriage return and line feed.
DISPLAY 27;15;5;10; Position cursor row 4 col 10
(vt100).
DISPLAY 27;15;5;10;"HERE"
R=5
C=10
DISPLAY 27;15;R;C;"HERE"
```

A/BASIC DISK INPUT/OUTPUT CONVENTIONS

In OS-9, A/BASIC uses the facilities of the host operating system, for input and output to and from disk files. In order for programs generated by A/BASIC to operate properly the I/O modules of OS9 must be memory resident.

Disk I/O in A/BASIC is channel oriented. This means that if your program references another file for input, output, or update, that file must first be "opened," and given a "channel number." A/BASIC supports up to 10 channels in addition to OS-9's 3 standard I/O paths.

All disk files are defined in accordance with OS9 and FLEX standards and are always ASCII files with the exception of 'CHAINED' files.

Many of the A/BASIC disk operations use OS9 and FLEX calls as would assembly language programs so information as to disk operations listed in both manuals will apply.

Note: In the descriptions that follow the term "filnum" refers to a channel number which is a constant in the range of 0 to 9.

OPEN Statement

SYNTAX: OPEN #filnum,strexp

This statement is used to open a file for update and assign the file to the channel number. One and only one file may be open on a particular channel at a time. The file must already exist or an error will be generated.

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

The file name specification is a string constant, variable or expression and must conform to OS9 and FLEX standards for device, directory and file name.

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

```
OPEN #0,"test"   Opens file test on channel 0
A$="/d1/master"
OPEN #2,A$       Opens /d1/master on channel 2
```

CREATE Statement

SYNTAX: CREATE #filnum,strexp

This statement works the same as OPEN except that the file is created and therefore must not already exist. After the file has been created it is left open in the update mode.

Note. A file must be opened with OPEN or CREATE before any I/O can be performed on it.

Examples:

OS-9:

```
CREATE #0,"test"
CREATE #2,"/D1/DEFS/special.stuff"
```

FLEX:

```
CREATE #1,"FILE.NAM",X
where X=number of 252 byte records.
```

CLOSE Statement

SYNTAX: CLOSE #filnum, ... ,#filnum

This statement is used to close a file that was previously OPENed or CREATEd after completion of the I/O operations. CLOSE frees up the channel number and makes it available for reassignment if desired. Note that more than one channel can be closed with one statement.

Examples:

```
CLOSE #2
CLOSE #1,#2,#3
```

NOTE: THE CLOSE Statement SHOULD BE USED ON ALL OPEN FILES BEFORE EXITING THE PROGRAM TO ENSURE OF PROPER UPDATES.

WRITE Statement

SYNTAX: WRITE #filnum,outlist

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

The output list is a sequence of string and or numeric constants, veriables or expressions seperated by commas. Each element in the list is considered to be an "item" within the disk record to be written. Strings are written to a maximum size of 32 characters (or there dimension) and numeric values are written in ASCII decimal form. This conversion is automatic and allows for compatability with most BASIC disk files. The resulting record length must be less than or equal to 128 bytes.

CONT.

WRITE Statement CONT.

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 #0,"MASTER FILE NUMBER "+STR$(N)
```

Results of writing a disk record:

Each item is written on the record with a comma as a separator. 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,-400,"WORD",N
```

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

32	35	2C	2D	34	30	30	2C	57	4F	52	44	2C	31	30	0D
2	5	,	-	4	0	0	,	W	O	R	D	,	1	0	EOR

RWRITE Statement

SYNTAX: RWRITE #filnum,position,outlist

The RWRITE command works the same as the WRITE command except that it performs a "seek" to "position" before executing the write.

Example:

```
L=50 : REM set L = to the record length
R=33 : REM set R for the desired record #
RWRITE #2,R*L,A$,A,B : REM write the record
```

READ Statement

SYNTAX: READ #filnum,varlist

The read statement causes the next record of the 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 and or string type which may be subscripted if desired.

The number of items given in the variable list should agree with the number of items in the disk data record (except as noted below). The file must be open for read or update and must be an ASCII text-type file. Disk records are of variable length and are compatible with files written by BASIC09 using 'WRITE'.

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 the list		
	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 the variables in the list, they will be ignored. Note that the 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.

RREAD Statement

SYNTAX: RREAD #filnum,record,varlist

The RREAD command works the same as the READ except that it performs a "seek" to "position" before executing the read.

Example:

```
RREAD #2,25,A$,A,B : REM read A$,A and B from position 25
L=50 : REM set L + to the record length
R=33 : REM set R for record # desired
RREAD #0,R*L,A$,A,B : REM read record 33
```

RESTORE Statement

SCRATCH Statement

SYNTAX:

```
RESTORE #filnum, ... ,#filnum
SCRATCH #filnum, ... ,#filnum
```

These statements are used to reposition a file open for update to the first byte of the file. Because of OS9s use of file handling (all files being open or created for update) the two statements are functionally equivalent. Note that RESTORE and SCRATCH by themselves do not alter a file in any way, they just 'reposition' the file.

Example:

```
SCRATCH #2,#3

RESTORE #0,#1,#2
```

KILL Statement

SYNTAX: KILL strexp

The file name specified in strexp is permanently deleted from the system. Use this statement with care as the file may not be recovered again.

Example:

```
KILL "TEMP3"

A$="/d0/workdir/temp.file" : KILL A$
```

ON EOF GOTO Statement

Syntax:

```
ON EOF GOTO <line #>
ON EOF
```

Description: This statement works the same as ON ERROR does except that it traps an end-of-file condition during a READ or RREAD operation. See ON ERROR for further discusion.

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 (FLEX only)

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 a 0.

Example: IF EOF (#4)<=1 THEN 550

ON EOF Statement

SYNTAX: ON EOF GOTO 100

This statement works the same as "ON ERROR " but traps an end of file condition.

RENAME Statement

SYNTAX: RENAME A$,B$

This will rename the file A$ to B$.

FILSIZ Function (FLEX only)

SYNTAX: FILSIZ (#filnum)

This function returns the current length of the file specified in sectors.

STATUS Function

SYNTAX: STATUS(#filnum)

Returns the current status of the file specified as follows:

0 - File not currently open
1 - File is open for read
2 - File is open for write
3 - File is open in update mode

Example: IF STATUS(#3)<>3 THEN 450

COMPILATION PROCEDURES for FLEX

The first step in producing an A/BASIC program is to use an editor to create the source file containing the A/BASIC program. A/BASIC accepts the file format of most FLEX compatible Text Editors.

The syntax for the compilation is:

ABASIC,Source[,Object][,+Options]

Source = input source file specification
Object = generated object file specification (optional)
+Options = compile time options
H = print generated code in Hex format
S = print symbol table at end of listing
L= do not generate a listing

Note: Specifying the Object file to have the same name as an already existing file will cause the existing file to be deleted.

COMPILATION PROCEDURES for OS-9

The first step in producing an A/BASIC program is to use the system's text editor to create the source file containing the A/BASIC program. A/BASIC accepts the file formats used by most OS9 compatible editors.

The syntax for calling the A/BASIC compiler is given below:

Syntax: ABASIC pathlist [options] [>pathlist]

pathlist = The pathlist for the input source file

options = Valid compiler options (see below)

>pathlist = The standard output path may be redirected so that the listing goes to a file/device other than the terminal.

The folowing compile time options are available.

H = Print the generated hex code in the listing (used for debugging).
S = Print a symbol table at the end of the listing.
L = Do not generate a listing.
O = pathlist = Generate an object file specified by pathlist. If no directory is defined in pathlist the current execution directory is assumed.

ERROR HANDLING

When A/BASIC detects an error in the source program during either the first or second pass it will print the source line in error and a message with an error code (the codes are listed in the appendix). The line below the erroneous source line will have an up arrow showing the approximate position of the error. The error location is about 95% accurate.

When an error is detected on a source line the compiler will not process the line further even if it is a multiple statement line, so the rest of the source line should be examined carefully for possible undetected errors.

The error count printed in the listing is the number of errors detected during the second pass only.

THE PROGRAM LISTING

During the second pass a formatted listing is produced which will include any error messages (see COMPILE TIME ERRORS). The listing usually consists of three fields: the first is the hex address where the object code starts for the BASIC source line; the second field is the statement line number if any; and the last field is the BASIC source line.

After compilation is complete, the program statistics, load map and symbol table (if the program included an OPT S statement) are printed. The load map lists the names and addresses (main entry point) of the BASLIB modules the compiler selected and included in a particular program. The BASLIB module names and functions are:

NAME	Function
OUTPUT *	Terminal output modual (PRINT, DISPLAY)
INPUT *	Terminal input modual (INPUT)
DISK *	Disk file support (OPEN, CLOSE, READ, RREAD, RWRITE, CREATE)
MULTIPLY	Multiplication (Arithmetic)
DIVIDE	Divide (Arithmetic)
RANDOM	Random number generator (RND)
STRING 1 *	String primitives and functions
STRING 2 *	Extended string functions (SUBSTR, VAL, STR)

* THESE MODULES HAVE MULTIPLE ENTRY POINTS.

IN 0

SYMBOL TABLE FORMAT

The symbol table listing has four columns. The first is the name of the variable, or one of four types of compiler-allocated variables:

```
*   = for/next loop limit/step value (2 bytes)
#n  = I/O control block (variable size)
IO  = Input/Output buffer (129 bytes)
ST  = string working buffer (256 bytes)
```

The second column is the hex address of the first (most significant) byte of the variable.

The last two columns indicate the size in hexadecimal variable as follows:

simple numeric	both 0
numeric array	# rows, # columns (0 for 1-dim.)
simple string	string length, 0
string array	string length, # of strings

APPENDIX A

A/BASIC RUN-TIME ENVIRONMENT

Memory Assignments:

A/BASIC programs should reserve addresses below #0030 for BASLIB temporary storage. The usage of this area of memory by A/BASIC is listed below:

$0000-$000F	SCRATCH AREA FOR FAST MATH (MUL,DIV,DSK)
$0010-$0011	FILE CONTROL BLOCK TEMP. ADDR. (DSK)
$0012-$0013	ERROR TRAP ADDRESS (ST2,DSK,MUL,DIV,INP)
$0014	ERROR TYPE CODE (ST2,DSK,MUL,DIV,INP)
$0015-$001F	NOT USED BUT RESERVED FOR FUTURE USE
$0020-$0021	XR TEMPORARY (INP,OUT,ST1,ST2)
$0022-$0023	I/O BUFFER POINTER (INP,OUT,DSK,ST2)
$0024-$0025	RESERVED
$0026	I/O BUFFER ZONE COUNTER (INP,OUT,DSK,ST2)
$0027-$0028	STRING BUFFER POINTER (ST1,ST2)
$0029-$002A	STRING XR TEMP (ST1,ST2)
$002B-$002C	STRING TEMP (ST1,ST2) MULTIPLY OVERFLOW HIGH-ORDER 16 BITS (MUL)
$002D-$002F	STRING FUNCTIONS TEMP (ST1,ST2)

APPENDIX B

A/BASIC LANGUAGE SUMMARY

ASSIGNMENT:

LET	POKE

CONTROL:

CALL	FOR/TO/STEP	GOTO	NEXT
GOSUB	RETURN	IF/THEN	IF/GOSUB
IF/THEN/ELSE	ON ERROR GOTO	ON EOF GOTO	ON OVR GOTO
ON NOVR GOTO	ON/GOTO	ON/GOSUB	STOP
CHAIN	SHELL		

INPUT/OUTPUT:

INPUT	PRINT	DISPLAY	OPEN
CLOSE	READ	RREAD	WRITE
RWRITE	CREATE	RESTORE	KILL
SCRATCH	RENAME		

COMPILER DIRECTIVES:

GEN	BASE	ORG	END
DIM	REM	'.'	'*'
PAGE	NAME		

NUMERIC FUNCTIONS:

ABS	POS	RND	PEEK
TAB	ASC	LEN	SUBSTR
VAL	ERR	EOF	STATUS
SWAP			

STRING FUNCTIONS:

CHR$	LEFT$	RIGHT$	MID$
STR$	TRM$	BUF$	

OPERATORS:

+	ADD	&	AND	+	CONCATENATE (STRING)
-	SUBTRACT	!	OR		
/	DIVIDE	%	EXCLUSIVE OR		
*	MULTIPLY	#	NOT		
-	NEGATE				

APPENDIX C

A/BASIC COMPILE-TIME ERROR CODES

02 Line number duplicated or out of sequence
03 Unrecognized statement
04 Syntax error
05 Variable name missing or in error
06 Equal sign missing
07 Undefined line reference: GOTO or GOSUB to nonexistant line number
08 Right parentheses missing or misnested
09 Operand missing in expression
10 Destination line number missing or in error
11 Number missing
12 Misnested FOR/NEXT loop(s)
13 Symbol table overflow *
14 Illegal task number - must be 0 to 15
15 Missing or illegal usage of relational operator(s)
16 Delimiter (, or ;) missing
17 Quote missing at end of string
18 Illegal type or missing counter variable in FOR statement
19 Redefined array
20 Error in array specification: subscript missing; too many subscripts
21 Error in array specification: subscript zero or larger than 255
22 Variable storage overflow, tried to allocate past $FFFF
23 Reference to undeclared array
24 Subscript error
25 Missing, illegal type or incorrect function argument (numeric)
26 Illegal option
27 Unrecognized operator in string expression
28 Concatenation operator (+) missing
29 Missing, illegal type or incorrect function argument (string)
30 Too many FOR/NEXT loops active - max is 16
31 Line reference table overflow *
32 Program storage overflow - tried to allocate past $FFFF
34 GOTO or GOSUB missing
35 Illegal channel number: must be 0 to 9
36 Error in disk I/O list

* These error types are not program errors. If more system memory is available the tables may be expanded to include more entries.

APPENDIX C

A/BASIC RUN-TIME ERROR CODES

The ERR function will return one of the following codes after an error occurs at run-time:

ERRORS 0 TO 31	DISK ERRORS The codes used are identical to those used by the host DOS and may be found in the system's DOS manual. All codes may not be implemented by DOS.
ERROR 32	MULTIPLY OVERFLOW The result of a multiplication exceeds the range +32767 to -32768. The result was the low order 16 bits of the result and the high order 16 bytes are saved in the fast scratch area.
ERROR 33	DIVIDE ERROR A divide with a zero divisor was attempted. A result of zero was returned.
ERROR 34	CONVERSION ERROR The BASLIB ASCII-to-binary conversion read illegal, oversize or no input. A value of zero was returned.

APPENDIX D

A/BASIC Reference Table

FUNCTIONS

ABS (n) absolute value of n

ASC (s$) numeric value of first character of s$

BUF$ I/O buffer

CHR$ (n) ASCII character corresponding to n

EOF (#n) test file n for end of file condition

ERR error number

FILSIZ (#n) number of sectors in file n

**LEFT$** (s$,n) string representing n characters starting at left of s$

LEN (s$) length in bytes of s$

**MID$** (s$,n[,n]) string representing n2 (or remaining) characters starting at nl characters into s$

PEEK (n) 8-bit numeric value at address n

POS character position in print buffer

**RIGHT$** (s$,n) string representing n characters at the end of s$

RND [(n)] random number between 0 and 255

STATUS (#n) status of file n

STR$ (n) string conversion of numeric n

SUBSTR (s$,s$) first occurrence of sl$ in s2$ (or zero if not found)

SWAP (n) swap bytes of 16-bit value n

TAB (n) advance print buffer pointer to position n

**TRM$** (s$) argument s$ without trailing spaces

VAL (s$) numeric conversion of string s$

APPENDIX D

A/BASIC Reference Table

OPERATIONS

! binary logical inclusive or (l!r)

" ... " unary string constant definition

unary logical complement (#r)

% binary logical exclusive or (l%r)

& binary logical and (l&r)

(unary expression group start

(# unary file number group start

) unary expression group end

* binary numeric multiply (l*r)

+ binary numeric addition (l+r),
binary string concatenation (l$ cat r$),
unary numeric positive (+r)

, binary subscript separator,
binary parameter separator,
binary PRINT punctuation (tab)

- binary numeric subtraction (l-r),
unary numeric negation (-r)

/ binary numeric division (l/r)

: binary separate statements on line

; binary PRINT punctuation (no tab),
unary PRINT punctuation (no CRLF)

< binary numeric less (l<r),
binary string less (l$<r$)

<= binary numeric not greater (l<=r),
binary string not greater (l$<=r$)

<> binary numeric not equal (l<>r),
binary string not equal (l$<>r$)

= binary numeric equal (l=r),
binary string equal (l$=r$)
binary numeric assignment (l=r),
binary string assignment (l$=r$)

=< binary numeric not greater (l=<r), binary string not greater (l$=<r$)

OPERATIONS cont.

=> binary numeric not less (l=>r),
binary string not less (l$=>r$)

> binary numeric greater (l>r),
binary string greater (l$>r$)

>= binary numeric not less (l>=r),
binary string not less (l$>=r$)

APPENDIX D

A/BASIC Reference Table

STATEMENT

***** ... introduce remark (column 1)

. ... introduce remark (column 1)

BASE [=]n set ram assignment address to n

CALL n call machine language subroutine at address n

CHAIN s$ load and run BASIC program named s$

CLOSE #n[,#n[,...]] close specified files or all files

CLOSE FILES close specified files or all files

CREATE #n,s$ create file n with name s$

DIM v(n[,n])[,...] declare dimensioned variables

DIM v$(n[,n])[,...] declare string variables with length n1 or string arrays of dimension n1, length n2

DISPLAY [e[,]][e[;]][,...] output strings and control characters to terminal

END terminate execution

EXPAND expand memory space

FOR v=n TO n [STEP n] create loop with control variable v set initially to n1, terminal condition of v crossing n2, step size n3 (or 1)

GEN N[,N[,...]] insert specified values into program

GOSUB N call subroutine starting at line N

GOTO N branch to line N

IF x GOSUB N call subroutine starting at line N if expression x true

IF x THEN N branch to line N if expression x true

IF x THEN S perform statement S if expression x true

INPUT L input list L from terminal

KILL s$ delete file named s$

STATEMENTS cont.

[LET] v[(n[,n])]=n assign expression on right of equal to variable on left

NAME S$ set module name to S$

NEXT v initiate next iteration for FOR loop with control variable v

ON ERROR GOTO [N] set trap at line N for error handling or terminate error handling trap

ON NOVR GOTO N branch to line N if no overflow

ON OVR GOTO N branch to line N if overflow

ON n GOSUB N[,N[,...]] call subroutine at n-th line number N

ON n GOTO N[,N[,...]] branch to n-th line number N

OPEN #n,s$ open file n1 with name s$

OPT L provide compile options

ORG [=]n set program address to n

PAG continue compiler listing on next page

POKE (n)=n store 8-bit value n2 at address n1

PRINT [e[,]][e[;]][,...] output characters to terminal

READ #n,L read data into list L from sequential file n

REM introduce remark

RENAME rename file A$ to B$

RESTORE #n[,#n[,...]] rewind files n1, n2, ... and reopen for input

RETURN return from most recent active GOSUB

RREAD #n,n,L read data into list L from random file n1 record n2

RWRITE #n,n,L write data from list L into random file n1 record n2

SCRATCH #n[,#n[,...]] rewind files n1, n2, ... and reopen for output

SHELL s$ pass message s$ to operating system

Even More STATEMENTS

STACK [=]n set initial stack pointer to address n

STOP terminate program execution

WRITE #n,L write data from list L into sequential file n