A/BASIC COMPILER

REFERENCE MANUAL

MICROWARE SYSTEMS CORPORATION
P.O. BOX 954 • DES MOINES, IOWA 50304 • (515) 279-9856

A/BASIC COMPILER REFERENCE MANUAL

Software License and Limited Warranty:

A/BASIC and RTEDIT are copyrighted property of Microware Systems Corporation and are furnished to its customers of use on a single computer owned by the customer. The programs may not be modified or copied except for use on the customer's computer system. This license is not transferable and the customer shall not make this software available to any third party without express written permission of Microware Systems Corporation.

Microware Systems Corporation warrants this software and manual to be free from defects for a period of 90 days after the date of purchase. Any corrections may be supplied to the customer in the form of printed material, magnetic tapes, or disks, at the option of Microware Systems Corp.

Microware Systems Corporation reserves the right to make changes without notice (except within the warranty period) and cannot be responsible for any damages (including indirect or consequential) caused by reliance on the performance of this software or the accuracy of its documentation.

Purchase of this material is considered implied consent of the provisions of this license and warranty, unless returned to Microware Systems Corporation with the media package seal unbroken within ten days of the date of purchase.

To our customers: Microware Systems Corp. is committed to providing high-performance, low cost software to you, now and in the future. This is not possible if unauthorized distribution and sale occurs. We ask your cooperation in controlling the distribution of this software to non-purchasers and notifying us of cases of unauthorized reproduction of this material so appropriate action may be taken. Thank you.

OVERVIEW OF A/BASIC

A/BASIC is a two-pass compiler for the M6800 family of microprocessors which converts programs written in a modified, extended BASIC language to pure M6800 machine-language programs.

A/BASIC is oriented towards applications currently programmed in assembly language, and is designed to generate optimized, compact and high-speed machine language. The output of the compiler can be run as stand-alone RAM, ROM or PROM programs, or multiprogrammed using RT/68 operating system's multitasking capabilities.

The compiler does not require use of any "run-time package" as it contains an internal subroutine library "BASLIB" which contains copies of all required run time-subroutines. Integral to A/BASIC is a subroutine linkage editor which places one and only one copy of any required subroutine in the machine language program.

The compiler requires a minimum of 8K bytes of memory which is sufficient to typically compile 200 to 600 line programs. If additional memory is available, it may be used to expand tables so programs of virtually any size may be compiled.

A/BASIC AND its host operating system, RT68, may be used to effectively create fast, efficient programs for applications in areas such as:

* REAL-TIME CONTROL
* TEXT AND WORD PROCESSING
* DATA AQUISITION AND ANALYSIS
* SYSTEMS PROGRAMMING
* DATA COMMUNICATIONS
* SIMULATIONS AND GAMES
* REAL-TIME GRAPHICS
* BUSINESS PROGRAMS

The A/BASIC compiler and the RT/68 operating system can be customized for OEM applications that require special extensions or different operating environments.

INTRODUCTION AND DEFINITIONS

Because A/BASIC is both unique and quite powerful when used to full capability, it is highly reccommended that this manual be studied carefully before compilation is attempted.

Programmers with primary experience using interpreters will find several major differences between compilers such as A/BASIC and . BASIC interpreters, mainly:

* There is no interactive operation during program debugging and testing. Often several compilations are required to produce the final version.

* While a machine language program is in execution, "run-time" errors are not detected unless checks were "written in" to the program, i.e., illegal input, array overflow, etc.

* Compilers, particularly A/BASIC, require the programmer to "lay out" memory assignments and utilization, and generally require a more sophisticated knowledge of actual machine characteristics and operation.

* Compiled machine langage programs run much faster and typically consume less memory than interpreted BASIC programs.

This manual is organized in such a way that it should be usable either for instuctional or reference purposes. In several areas references are made to other sections where a subject is more directly applicable.

Syntactic descriptions shown generally follow the form where a part of the syntax commonly used is enclosed such as <this>.

Commonly used definitions are:

<address> - a memory address represented as a decimal or hexa-decimal number. "Hex" numbers are predeced by a dollar sign such as $BF00.
<var> - A variable name. Usually the description that follows will qualify it as to what type it may be (string and/or numeric, etc.)
<expr> - An expression that results in a NUMERIC result, though it may include functions that have string-type arguments. An expression may be complex and contain many terms or it may be a single constant number or a single variable.
<line #> - A legal line number that occurs somewhere in the BASIC program.
<str expr> - An expression that evaluates to a STRING result even though it may contain functions which have numeric expressions for arguments.
<str var> - A legal string variable name, or an indexed (subscripted) string variable.

Others such as <option>, <IO spec>, <array spec>, etc. are defined where they occur.

A/BASIC GLOSSARY

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 absolute addresses 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 RT/68 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 pro-gam 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 constuction 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.

A/BASIC PROGRAM STRUCTURE

An A/BASIC program consists of one or more source lines. Each source line must begin with a line number which is a positive number of up to 4 digits. Line numbers on succesive statements must be in ascending order without duplication.

A source line may contain one or more BASIC statements. If more than one statement is included on a line a colon : character is used to seperate them. A line may contain up to 72 characters. Blanks (spaces) may be used to improve readability but are not significant to the compiler except when included in character strings enclosed in quotes.

Examples of BASIC source lines:

```
100 A=B+C : D=E*F
200 GOTO 1020
300 N=N+1 : GOSUB 520 : IF A<B THEN 200
```

Basic Statements

A/BASIC statements are grouped into five catesories:

ASSIGNMENT STATEMENTS are used to assign a variable a value which is the result of an evaluation of an expresssion.

CONTROL STATEMENTS are used to alter the normal sequence of program execution, which may be dependent on some condition.

INPUT/OUTPUT STATEMENTS are used to transmit and accept data from peripheral input/output devices.

REAL-TIME AND SYSTEM CONTROL STATEMENTS allow the program to interact with the system environment and RT/68 operating system.

COMPILER DIRECTIVE STATEMENTS do not directly produce any corresponding machine language, but rather control the compilation process and compiler operation.

A section of this manual is devoted to each catesory listed above in which detailed definitions of each statement appear.

ARITHMETIC OPERATIONS

Numbers

A/BASIC's numeric data type is internally represented as 16-bit (2 byte) 2's compliment integers. Therefore the basic range in decimal of a number is -32768 to +32767.

A negative number is preceded by the minus sign. Numbers may also be used in hexadecimal form in which case the hex digits are preceded by a dollar sign (not to be confused with strings!) and the range in hex is therefore -$8000 (-32768 decimal) to $7FFF (32767 decimal).

Because A/BASIC supports logical operations and memory access that may require 16 bit UNSIGNED integers it will also consider decimal numbers up to 65535 and hex $FFFF to be legal - it is the context that is important.

Examples of legal numbers:

```
    200   -5000  $0100  $FE00 -$200  $C  -1
```

Some ILLEGAL numbers:

    99999 (too big) -$FFFF(out of range) +20 (plus sign illegal)

Because of the way 2's compliment numbers are represented, the numbers -1, $FFFF, and 65535 all have the same binary representation.

Variables

A numeric variable in A/BASIC has a name which consists of a single letter or a single letter and a digit. Two bytes of RAM are allocated for each variable.

Examples of legal variable names:

```
  X  N  R2  A9  Z3  F
```

Arrays

Numeric array may have one or two dimensions (subscripts). The maximum subscript size is 255. Array names are the same as variable names. Before an array variable is used, it must be declared by means of the DIM statement.

When array variables are used, the subscripts may be expressions but must not be: negative, zero, or greater than 255. The object program does NOT check subscripts when they are calculated so if they are out of legal range they may destroy other data or the program.

Wherever possible use one-dimensional arrays as they may be accessed fastest. Though finding an address of a two-dimensional array requires a multiplication, A/BASIC will use a special fast 8-bit multiplication subroutine.

When a variable is declared as an array but used in the program without subscripts, it will be implied that it is the first element of the array. For example, if an array X(20,20) is declared, the compiler will interpret X+X to mean X(1,1)+X(1,1).

Examples of legal usage of arrays:
```
   X(N%2)   V4(3,N)  A(B(N+1))  S5(2,6)  W(N%2,(A+B)%C)
```

Arithmetic Operators

There are five legal arithmetic operators which may be used in arithmetic expressions:

```
+ add
- subtract
* multiply
/ divide
- (unary) negative
```

Arithmetic operations may produce errors due to overflow and underflow which may be tested (see EXTENDED ARITHMETIC OPERATIONS).

Logical Operators

The four logical operators perform bit-by-bit boolean functions on their operands and are useful for bit manipulation and testing, masking, etc. They may be mixed in numeric expressions with arithmetic operators as required.

```
@ - AND
! - OR
% - EXCLUSIVE OR
# - (UNARY) NOT
```

Order of Operations

Arithmetic expressions are evaluated on the basis of operator precedence where operators are grouped into priority classes. As the expression is scanned from left to right by the compiler, the expression will be evaluated by processing operators with the highest priority first, etc. Parentheses may be used to alter the order of evaluation.

```
1  FUNCTIONS
2  UNARY - AND #
3  @ ! %
4  * /
5  + -
```

Arithmetic Expressions

Arithmetic expressions may range from a single constant or variable to a long formula with many operators and operands. The compiler will attempt to rearrange the expression for optimum code generation wherever possible. If temporary storage of intermediate results is required, the machine stack is used. The compiler will generate the shortest instruction sequence when array references and function calls follow the natural order of operations.

Examples of legal expressions:

```
A   25 .. A*B   C+(D-F)  $200*N  C+(D@F*-$200/(RND/2)!N(X,Y))

A%(B@$FF00)  2*M3   A(N+1)+A(N+2)* B(N,M)  #$40*-100
```

Arithmetic Functions

The followins arithmetic functions are supported by A/BASIC:

ABS(<expr>) - absolute value of <expr>
POS - current position of output pointer in I/O buffer
CLK - RT/68 real time clock current time
PEEK(<address>) - contents of memory byte at <address>
RND - Random number from 0 to 32767. If a smaller range is desired, divide RND by largest number desired+1 For example, to obtain a random number 0<= N < 10 use (RND/10).
If an expression in parentheses is used with RND the value of the expression will be used as a new random number "seed".

Several functions that use string arguments return numeric values. See the section on STRING FUNCTIONS.

Extended Arithmetic Operations

A/BASIC has statements that allow error detection and recover for arithmetic error as well as permitting multiple-precision arithmetic. An overflow resultins from a multiply or addition will set the C (carry) bit in the MPU condition code register. An underflow from a subtraction (borrow) or attempt to divide with a zero divisor will produce the same effect. Two statement which are identical in operation may be used to detect these circumstances:

```
ON ERROR GOTO <line #>      and
ON OVR GOTO <line #>
```

These statements will jump to the line number specified if the carry bit is set AS A RESULT OF THE LAST ARITHMETIC OPERATION of an expression. If more than one operation must be tested, they may be written as successive expressions.

A statement which has the opposite effect, i.e., jump if NO overflow/error occured is:

```
ON NOVR GOTO <line #>
```

Multiple-Precision Arithmetic

If numbers greater than the 16-bit range of A/BASIC's numeric type must be processed, two or more 2-byte segments of larger numbers may be stored, operated upon, and tested by treating several A/BASIC variables as one larger number. In this way operations may be performed on numbers represented as multiples of 16 bits, i.e., 32, 48, 64 ... bits.

For addition and subtraction, the ON OVR GOTO and ON NOVR GOTO statement may be utilized. For example, assume two 32 bit numbers are stored in variable pairs A1, A2 and B1, B2 respectively. A1 and B1 shall be the variables containing the most significant halves of each value. The following subroutine may be used to add them:

```
300 A2=A2+B2 : ON NOVR GOTO 310 : A1=A1+1
310 A1=A1+B1 : RETURN
```

To subtract B from A the following subroutine is used:

```
400 A2=A2-B2 : ON NOVR GOTO 410 : A1=A1-1
410 A1=A1-B1 : RETURN
```

Similar routines may be written for multiplication and division. If an overflow occurs on a multiply, the multiply subroutine will return the least significant 16 bits of the result and place the high-order 16 bits of the product in the fast scratchpad area at addresses $002B and $002C.

The division routine does not preserve the remainder, if any, but because of the integer arithmetic the remainder of A/B can be found by evaluating A-A*(A/B).

STRING PROCESSING

Character Strings

The strings data type is used for operations that involve processing text represented as variable length strings of characters. The data type allows use of all characters in the ASCII code except the NULL character ($00) which is used as an internal end-of-string delimiter.

String Constants

A string constant consists of any number of characters enclosed in double quote (") symbols. It may not include the carriage return symbol ($0D). If a double quote is to be part of the string, two adjacent to each other will be interpreted as one double quote and not end-of-string. Spaces within a string are considered characters. Examples:

```
"A VERY SHORT STRING"
"A DOUBLE QUOTE "" WITHIN"   (read as A DOUBLE QUOTE " WITHIN)
```

String Variables

String variable names consist of a single letter and a dollar sign so there are 26 possible string variable names. One-dimensional string arrays may be declared using the DIM statement. All subscripting rules for numeric-type arrays also are applicable to string arrays, including the characteristic that a string variable name declared as an array but used without a subscript is assumed to refer to the first element. For example, if an array A$(10) is declared and A$ is referenced without a subscript, the compiler will assume A$(1).

A string variable is allocated 32 bytes of RAM so it may contain up to 32 characters. The data is internally represented as left-justified (first character at lowest address) with a zero-byte terminator if less than 32 characters long.

String Expressions and Concatenation

When string variables are used in an A/BASIC program the compiler will allocate a minimum of 256 bytes of RAM for use as a string operations buffer. When string operations are performed the strings are moved to and operated on in this buffer. Because the string buffer is the last RAM allocated by the compiler if more memory is available beyond 256 bytes it can automatically be used for additional string working storage.

The basic operator in string expressions is the + symbol which represents concatenation. This essentially is the function of joining the left string (constant, variable or function) with the right string (constant, variable, or function) to form one string.

Several strings may be joined in an expression by using + operators in the expression. While processing a string expression, the string buffer typically fluctuates in size according to how long intermediate results of the evaluation are.

String expressions are evaluated from left to right, and no parentheses or operators other than + or functions are permitted.

The result of a string evaluation leaves the string result in the buffer which can be moved into memory in the case of a string assignment statement. If the result is to be stored as a string variable, up to the first (left) 32 characters will be stored. In the case of the special string variable BUF$, up to 128 characters will be stored.

String Functions

There are ten string functions grouped into two classes. The first class return a NUMERIC VALUE and have as arguments string expressions. The second class return STRING VALUES and have as arguments string and/or numeric expressions. The second class are identified by having a dollar sign as part of the function name.

Class 1 Functions -

ASC(<str expr>) - returns a number which is the numeric ASCII code for the first character of the string expression. An empty string returns 0.

LEN(<str expr>) - returns the length of the string expression up to 255 characters. An empty string returns 0.

SUBSTR(<str expr>,<str expr>) - searches for the first occurance of the first string within the second string, in other words, searches for the first string in the second string. If it is not found, a value of 0 is returned. If it is found, the number returned is the character position whre the beginning of the substring was found. For example:

```
SUBSTR("EXAM","AN EXAMPLE")  will return 4
if A$="CAT" then  SUBSTR(A$,"ANIMALS") will return zero
```

VAL(<str expr>) - returns the numeric representation of a number represented as a string of ASCII characters, in other words, a string to numeric conversion. The conversion starts at the beginning of the strings, and leading spaces are skipped. The number may be positive or negative in the ranse -32768 or +65535. The conversion ends when 5 digits are read, or a blank, comma or end-of-string are reached. If the string is not ASCII symbols for digits or any error occurs, a value of zero is returned. Examples:

```
VAL("123,456") returns 123 numeric
VAL("-10000000)  returns zero
VAL("TEN") returns zero
```

Class 2 String Functions:

CHR$(<expr>) - returns the ASCII character corresponding to the number which is the result of the evaluation of the numeric expression.

LEFT$(<str expr>,<expr>) - returns the leftmost N characters of string expression. If the length of the string is less than N the entire string is returned. (N is the result of evaluation of <expr>).

MID$(<str expr>,<expr 1>,<expr 2>) - returns the middle part of the string <str expr> beginning from character position <expr 1> and including <expr 2> characters from that point. If <expr> is greater than the number of characters in <str expr> or <expr 2> is <= 0 a null string is returned. If there are not enough characters in <str exp> to satisfy the function, all characters from <expr 1> to the end are returned.
Examples:
MID$("DEMONSTRATION",3,5) returns "MONST"
MID$("LITTLE",10,5) returns ""
MID$("TOO SHORT",5,20) returns "SHORT"

RIGHT$(<str expr>,<expr>) - returns the right <expr> characters of the string <str expr>. If <expr> is greater than the number of characters in the string, the entire string is returned.
Example:
RIGHT$("DEMONSTRATION",4) returns "TION"

STR$(<expr>) - returns a string consisting of the ASCII characters that represent the numeric-to-string conversion of <expr>. If <expr> is negative a leading minus sign is included.

TRM$(<str expr>) - returns the string argument less any trailing blanks (spaces).
Example:
TRM$("SPACES ") returns "SPACES"

NOTE - IN ALL STRING FUNCTIONS THAT HAVE A NUMERIC ARGUMENT(S) EXCEPT STR$ THE VALUE OF <expr> MUST BE IN THE RANGE 0 TO 255. THE STRING ARGUMENT(S) MAY NOT EVALUATE TO A STRING WITH MORE THAN 255 CHARACTERS.

Null Strings

An important part of string processing involves the use of null or "empty" strings - strings constants or variables that contain no characters. This is represented as "". To initialize a sting variable, an assignment A$="" is analogous to the numeric operation A=0. This is important because strings are variable length with end of strings terminated by a zero byte. A/BASIC does not initialize variables to zero or null so initialization of certain ones may be required, depending on the program.

String Operations on the Input/Output Buffer

A very powerful feature of A/BASIC's string operations is the ability to perform string manipulations on the 128-byte input/output buffer. The special string variable BUF$ may be used anywhere a string variable is legally used. Additionally, special forms of the four input/output statements are available to read or write the buffer:

```
PRINT BUF$
INPUT BUF$
TREAD BUF$
TWRITE BUF$
```

Any of the above statements will read/write up to 128 characters to/from the terminal or cassette system and deposit them in the input/output buffer BUF$ without any change. Also, user-written subroutines in A/BASIC or machine language may read or write data to/from BUF$ to devices not supported by A/BASIC directly, such as disks, tape devices, memory-mapped video displays, etc. BUF$ is also useful for formatting data or reading unformatted data when standard BASIC I/O statements will not operate properly, such as reading strings that have imbedded commas, etc.

Note that BUF$ is not the same buffer as the string buffer so references to BUF$ in expressions do not disturb data in it. Though BUF$ is used as a string variable, it is not limited to 32 character string lengths - the entire 128 byte buffer is used.

Data in BUF$ is modified only when:
1) another input/output statement is executed
2) BUF$ is given as the result of a string assignment statement, for example: BUF$=A$+RIGHT$(B$,N)

Examples of BUF$ Usage:

Adding a string to the end of the buffer-
```
BUF$=BUF$+A$
```
Removing the last character from BUF$-
```
BUF$=LEFT$(BUF$,LEN(BUF$)-1)
```

Read a line from the terminal and print it backwards:

```
100 INPUT BUF$
200 N=LEN(BUF$)
300 FOR K=1 TO N
500 BUF$=RIGHT$(BUF$,1)+MID$(BUF$,2,N-1)
600 NEXT K
700 PRINT BUF$
```

String Compare Statement

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

Description: The first string expression is tested for equivalence against the second string expresssion. For the two strings expresssions to match, they must be of the same length and include the same characters exactly including spaces.

Two string relationals are permitted: = equal to and <> or >< not equal to. Use of any other relationals will result in an error. If the relation is true, the line number specified will be executed. Examples:

```
IF A$=B$ THEN 200
IF A$(N+1)+"END"<>C$+LEFT$(B$,5) THEN 1040
IF A$ = "MATCH" THEN 2305
```

Code Generation for String Operations

If string operations are used by a program, the linker/editor will select string subroutines in BASLIB for inclusion in the program. For simple string assignments such as A$=B$ the variables are transfered directly from memory to memory and never enter the string buffer. When string operations that involve one or more levels of pending operation (such as LEFT$, MID$, etc.) the string values are moved to the string buffer as they are encountered in the source line. Pointers to the beginning position of each unprocessed string value are pushed onto the system stack when entered and pulled when ready to process, so the string buffer acts as a "string stack". The string buffer may grow to a maximum size when many string operations are pending such as:

```
LEFT$(A$+RIGHT$(B$+MID$(C$+D$,N,M),X),Y)
```

In this example, A$,B$,C$ and D$ will be moved to the string buffer at the point the MID$ operation is finally processed. This would require 32*4 or 128 bytes of buffer to store all strings, worst case. Because the compiler allocates a 256 byte string buffer, almost any expression can be accomodated EXCEPT when several operations are pending which use BUF$ for an operand, which may contai up to 128 bytes. In such a case it is desireable to use multiple statements for BUF$ operations so never move than 1 is pending OR allow extra room at the end of variable storage for the string buffer to expand.

Calculation of addresses of string variables which are arrays requires a multiplication. To generate code to address string array elements, the linker will edit in subroutine calls to the BASLIB fast 8 by 8 register multiply subroutine which can calculate the absolute address at run-time in less than 200 MPU cycles. (The same subroutine is used for 2-dimension numeric arrays).

ASSIGNMENT STATEMENTS

Arithmetic Assignment Statements

Syntax: LET <var> = <expr>
<var> = <expr>
POKE <address> = <expr>

Description: The arithmetic assignment statement evaluates the expression <expr> and places the result in <var> which may be a simple numeric variable or an indexed numeric variable, in which case the subscript(s) may also be expressions. Use of the LET keyword is optional and in no way affects the operation.

If the assignment statement is specified as a POKE, the low-order byte of the result is stored at <address>. The high-order byte is ignored. Note - the two MPU accumulators ACCA and ACCB always contain the result of the expression (ACCA is MS byte) after execution of the statement. If the result destination was indexed, the address of the array element will be in the MPU index register. This information may be useful when calling machine language routine where numeric data is to be passed.

String Assignment Statement

Syntax: <str var> = <str expr>

Description: The string expression is evaluated, the result of which is the final contents of the string buffer. The leftmost 32 characters (or less, if the result is less than 32 characters) are moved to <str var> which may be simple or indexed.

A special form of the string assignment, BUF$=<str expr> is described in the STRING OPERATIONS section.

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
```

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. At compile time, up to 16 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 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 JSR and RTS machine instructions, respectively.

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. For operation with strings, see the STRING PROCESSING section. Legal relations are:

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

On Condition Statements

See ARITHMETIC OPERATIONS section.

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 or equal to zero, or is higher than the number of line numbers 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 by causing a jump to the entry point of the RT/68 console monitor. The END statement should not be confused with STOP as STOP will not terminate compilation of the program.

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.

Character level input/output subroutines used are those provided by the RT/68 I/O system. Vectored I/O through an ACIA-type interface at any address may be accomplished by storing the address of the ACIA at RT/68's vector location. Refer to the RT/68 systems manual for details.

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 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 carriage return/line feed is printed when the last character in input.

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, up to 32 characters from the current position will be accepted including blanks, if any. The string field is terminated by a comma or end of line, or when 32 characters are read. If no characters are available, a null string will be returned.

Examples:

```
INPUT A,B,S$,B$
INPUT A(N+1,M-1),R,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 expresssion 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 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 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$;TAB(N+M);B$
```

Tape Input/output

Syntax: TREAD <IO list>
TWRITE <out spec><delimiter> ... <delimiter><out spec>

Description: The tape I/O statements are used to write data records on audio cassettes. The format, syntax, and functions of TREAD and TWRITE are identical to INPUT and PRINT respectively in regards to the way data is placed in/extracted from the I/O buffer. The major difference is in the way the record is written to or read from tape.

The tape record format is:
<90 NULLS><$02><UP TO 128 DATA BYTES><$03>

During execution of TREAD, a READER ON control character is transmitted and a loop is entered to ignore characters until a $02 start-of-record character is read. All characters following are read and placed in the input buffer until either 128 bytes have been read or a $03 end-of-record character is read. A READER OFF control character is transmitted and the input buffer then processed in a manner identical to the INPUT statement.

The TWRITE statement fills the I/O buffer in a manner identical to the PRINT statement. A PUNCH ON control character is transmitted, then a leader of 90 nulls to allow for motor on/off time, a $02 start-of-record character, the up to 128 character contents of the I/O buffer, and the $03 end-of-record character followed by a PUNCH OFF control character.

Base Statement

Syntax: BASE = <address>

Description: The A/BASIC has two internal "pointers" used during compilation to control memory allocation: a "data" pointer used to keep track of the next available RAM location for assignment of storage for variables and arrays; and a "program counter" used to locate the address of the current machine instruction being generated. The BASE statement is used to set or change the data pointer. The ORG statement controls the program counter and is described below.

These two statements are critical for correct execution of the object program because they define where data (variables and arrays) are stored and where the program is stored. These two areas MUST NOT ever overlap.

During compilation, RAM storage is assigned for variables at the time they are first encountered in the program. The data pointer is used for the address for the variable, then the size of the storage required is added to it. Two bytes are required for each numeric-type variable or each element of a numeric array, and thirty-two bytes for each string variable or element of a string array.

After the last statement of the program has been compiled, the data pointer is used to allocate storage for a 129-byte input/output buffer (if the program uses input/output statements) and/or a 256 byte string buffer (if the program uses string operations). Though a 256-byte minimum string buffer is specified, it may actually be larger (see STRING OPERATIONS).

At the beginning of compilation, the data pointer is assigned an initial default value of $0030. Memory below this address is reserved for operating systems and run-time fast scratch storage.

Because more than one BASE statements may be used in an A/BASIC program, it may be used to specify RAM variable storage blocks, assign specific variable names to specific addresses (which may even be addresses of peripheral control and data registers.

IMPORTANT - Because the compiler will select the 6800 direct addresssing mode (2 byte instructions) wherever possible, numeric variables should be allocated storage at addresses $0030 through $00FF whenever possible. String variables and arrays may be located elsewhere. This technique may reduce program size by as much as 40 percent!

This is most easily accomplished by using a BASE statement at the beginning of a program to set the data pointer to a "high" memory address, then using DIM statements to declare all arrays and string variables or other variables to be assigned specific addresses, and then using the BASE statement again to set the data pointer to "low" memory.

Note that because A/BASIC considers variable X to be the same as array element X(1) the DIM statement may be used to declare "simple" variables by defining them as one-element arrays.

Where programs are long and many variables are used, "trial" BASE values may be used and the program compiled with the OPT S option selected so the symbol table is printed with the exact variable storage assignments. The exact values for the BASE statements may used in the next compilation.

In the following example, several uses of the BASE statement are shown. The example program is set up to:

1) Assign storage for variables reference in the program from addresses $0030 to $00FF.
2) Assign array and string storage above $0100
3) Assign the variable name P1 to a peripheral register at $8010
4) Assign the variable name A1 to a peripheral register at $8020

```
050 OPT S : REM ENABLE SYMBOL TABLE

100 BASE = $0100
200 DIM X(10),Y(20,4),Z3(4,100) : REM DECLARE NUMERIC ARRAYS
300 DIM S$(20),T$(10),U$(4) : REM DECLARE STRING ARRAYS
400 DIM A$(1),B$(1),C$(1),D$(1) : REM "SIMPLE" STRING VARIABLES
500 BASE = $8010 : DIM P1(1) : REM DECLARE PERIPHERAL
600 BASE = $8020 : DIM A1(1) : REM DECLARE PERIPHERAL
700 BASE = $0030 : REM NORMAL ALLOCATION STARTS HERE
800 REM REMAINDER OF PROGRAM FOLLOWS .....
```

Dimension Statement

Syntax: DIM <array spec> ... ,<array spec>

Description: This statement is used to declare arrays. All arrays must be declared before they are referenced in the program. Arrays may not be redefined. The maximum size for a subscript is 255. One or two-dimensional numeric arrays are permitted, and one dimensional string arrays may be declared. Each element of a numeric array requires two bytes of RAM storage, and each element of a string array requires thirty-two bytes of storage.

The DIM statement may be used in conjunction with the BASE statement to declare "simple" variables, or allow the same memory to be "shared" by different sized or named arrays.

```
Examples:  DIM A(10),B(3,8),N(255),B3(40)
           DIM B$(5),T$(100)
           DIM A(4),A$(4)
           DIM X($20)
```

End Statement

Syntax: END

The END statement is used to indicate the last statement of the source program. Compilation ceases for the pass when the END is encountered. Only one END statement is permitted, and it must be the last statement in the program. If a jump to the RT/68 monitor is desired at the end of the program, a STOP statement should precede the END. If end-of-file is read from the source input device before an END is encountered, the compiler will assume an END.

Options Statement

Syntax: OPT <option>, ..., <option>

Description: The OPT statement is used to activate options which affect the compilation process. The option symbols and functions are:

I - Inhibit generation of object file
H - Print hexadecimal instruction field in listing
N - No listing
S - Print symbol table

All options are not selected by default. If OPT H is selected, the compiler will print one or more lines of three columns of hex data after each source line. Each of these lines correspond to a machine instruction generated by the compiler. The leftmost column is the absolute address, the next column is the opcode, and the rightmost column is the operand (if any).

If the S option is selected, a dump of the compiler's symbol table is printed after the end of the second pass. The leftmost column is the symbol, the next columns are the absolute memory address, array row length (if array) and array column length (if array), respectively. If applicable, several "internal" symbols will be included to represent compiler-generated storage assignments. They are:

IO - Input/output buffer (129 bytes)
RR - Random number temp. (2 bytes)
ST - String buffer (256+ bytes)
ZZ - Index calculation scratch (2 bytes)
Any symbol with an asterisk - for/next loop terminate or increment temporary storage (2 bytes)

Examples: OPT S
OPT I,N,S

Program Origin Statement

Syntax: ORG = <address>

Description: The ORG statement is used to assign a value to the compilers internal "program counter" which is the address at which the compiler will generate the next instruction. This statement may be used more than once if it is desired to put different portions of the program at different memory areas. If used at the start of a program, it establishes the first address of the program. If no ORG statement is included, a default value of $1000 is supplied by the compiler which is the normal starting address.

It is extremely important that the program counter does not ever cause instructions to be located at areas assigned for variable storage. Note that if default values for BASE and ORG are used, a little under 4K bytes of storage for variables is available before any overlap occurs. See the BASE statement description for more information.

Note that if the ORG is used in the middle of a program to change the program counter (as might be required if a PROM boundary is reached, for example) a GOTO may be required before the ORG to jump to the succeeding program segment.

Examples: ORG=$7000
ORG = 8192

Remark Statement

Syntax: REM <text>

Description: This statement is used to include comments in the program. On multiple statement lines any REM must be the last statement on the line, or following statements will be ignored. Use of REM statements will not increase the length of the object program generated.

Examples: REM A COMMENT GOES HERE
REMARKABLE PROGRAMMING!

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)

IRQ On/Off Statements

Syntax: IRQ ON
IRQ OFF

Description: These statements are used to control the state of the MPU interrupt mask flag in the condition code register and are used to enable/disable interrupt recognition. These statements correspond directly to the 6800 CLI and SEI instructions. Refer to the RT/68 Systems Manual and the M6800 Programming guide for specifics on interrupt processing.

On Interrupt Statements

Syntax: ON IRQ GOTO <line #>
ON NMI GOTO <line #>

Description: These statements are used for generating programs to be used in the RT/68 SINGLE TASK MODE (non-multiprogramming) where interrupts are processed as vectors to specific service routines. When encountered in a program these statements cause the absolute address of the program corresponding to the line number specified to be stored at the interrupt vector addresses in the operating system scratchpad ($A000 for IRQ and $A006 for NMI).

The line number specified should be the beginning of the interrupt service routine which would typically service the device causing the interrupt. This routine is similar to a BASIC subroutine except it is terminated by an RETI (return from interrupt) statement instead of a RETURN statement.

Examples: ON IRQ GOTO 2010
ON NMI GOTO 400

Interrupt Return Statement

Syntax: RETI

Description: This statement terminates an interrupt-caused routine by loading the MPU register contents prior to the interrupt from the machine stack and resuming program execution from the point where the interrupt was acknowledged. This statement corresponds directly to the machine language RTI instruction.

Stack Assignment Statement

Syntax: STACK = <address>

Description: This statement is used to initialize or change the MPU stack pointer register. This is typically one of the first statements in an A/BASIC program if used. If a STACK statement if not included in the program, the operating system scratchpad stack ($A049 to $A016) will be used by the program for the machine stack. This is adequate for most programs that do not: nest subroutines extensively; use interrupts; run in the multiprogramming mode; or process elaborate arithmetic expressions. Otherwise, a specific memory area should be dedicated for the stack and the STACK instruction used to load the stack pointer with the TOP (highest address) of the stack.
Example: STACK = $A042

Switch Statement

Syntax: SWITCH

Description: This statement is used in the RT/68 MULTITASK MODE to call the executive, thereby causing the task to be suspended while other tasks may run. The task selection/switching process is described in section 4 of the RT/68 Systems Manual. The machine instructions generated by this statement include operations that set the MPU interrupt mask and the RT/68 task switch flag byte (RELFLG) and a software interrupt (SWI) which activates the executive. When the task is run again, execution resumes at the statement following the SWITCH statement.

Task On/Off Statements

Syntax: TASK <task #> ON
TASK <task #> OFF

Description: When used in the RT/68 MULTITASK MODE this statement will find the task status byte of the specified task in the status table and either set or clear the task state bit. This causes the task to be either activated (able to run) or disactivated (may not run) when the executive is selecting a task to execute. Note that this statement does not directly cause the task to start/stop- it merely enables/disables it from consideration when the executive is searching for a task to run. A task may use the TASK N OFF to turn itself off and become dormant. This is often used when the task was activated by another to perform a specific function and has completed its operation.

COMPILER OPERATION

PASS 1: During the first pass the entire source program is compiled in a "dummy compile" that performs a complete syntax analysis and pseudo-code generation to build the symbol table and line reference table. At the end of pass one the addresses will have been fixed in the line number table and symbol table.

As the first pass is performed references to subroutine calls to BASLIB routines are noted by setting individual flags for each routine required. At the end of pass 1 the linker/editor will allocate storage for all required subroutines at the end of the main object program, and allocate variable storage for any temporary variables or buffers required. These addresses are therefore "fixed" at the end of the first pass.

PASS 2: During the second pass the code generation system will produce object code. All addresses of forward and backward references are obtained from the line reference table, and subroutine (BASLIB) addresses and buffer addresses are obtained from the linker system. After the main object program is complete, the linker is called again to copy images of subroutines selected from BASLIB and append them to the end of the object program. Note that all routines contributed by BASLIB are "packed" at the end of the object program and one and only one copy of each is included.

Compilation Techniques: A/BASIC is extremely short primarily due to use of a sophisticated integrated scanner/parser/code generator that produces object code directly from source lines. Intermediate code is generated only for string and numeric expressions for optimization purposes. The basic parsing technique is a top-down, predictive system.

Code Generation: All code generation is designed to generate the fastest, shortest code possible. Instruction sequences and conventions take maximum advantage of the 6800 architecture and addressing modes. The machine stack is used extensively for fast intermediate storage. Arithmetic processing uses the MPU accumulators and hardware stack as a single virtual stack for optimum code generation. The arithmetic parser will attempt to rearrange expressions for shortest instruction sequences wherever possible.

The compiler will automatically select shorter direct addressing mode instructions wherever possible, and makes extensive use of short immediate mode instructions. The index resister is used as an operand pointer, secondary stack pointer, strins pointer, and for indirect addressing for arrays.

COMPILER OPERATIONAL PROCEDURES

The first time the compiler is to be used, it should be loaded and any modifications (SEE CUSTOMIZING A/BASIC) required for the specific system should be made. One or more backup copies of the compiler should be made by using the RT/68 DUMP command:

```
D,0100,1BBF
```

Preparing an A/BASIC Program

The RTEDIT program on side 2 of the cassette is used to create/modify an A/BASIC program. It is loaded using the RT/68 LOAD function and started by entering:

```
E,0100
```

It will print a question mark prompt and the program may be entered/edited. When the program is complete, it should be written on a tape using the "SAVE" or "END" functions. The editor is then exited.

The next step is to load the compiler program (side 1 of the cassette) using the RT/68 LOAD function. After the compiler is loaded, the tape (source file) is placed in the READ cassette machine and the cassette interface prepared for automatic read operation.

The compiler may be started by the RT/68 command:

```
E,0100
```

and should immediately start reading the source cassette. Nothing will be printed during this first pass except error messages. At the end of the first pass, the compiler will stop reading the tape and print the message "PASS 2". If any error occured during pass 1, no further compilation should be attempted because the object program will be incorrect. The source file should be re-edited to change lines in error.

The source tape should be rewound and prepared to be read again. If an object tape is to be made, it should be placed in the RECORD cassette machine and advanced past the leader. The cassette interface should be prepared for automatic record and read. When both tapes are set up properly, enter any character on the terminal and the compiler will print a page heading and start to read the source tape. Each line will be printed and object records written on the object tape as required.

After the last source line is processed, the A/BASIC linker/editor program will append any subroutines required by the program to the end of the object program (without interrecord gaps).

Program statistics are then printed:

ERRORS - are the (decimal) number of errors occuring on the second pass
VAR LEN - is the (hex) number of bytes of variable storage allocated by the compiler
PGM LEN - is the (hex) length of the object program in bytes
PGM END - is the (hex) address of the last instruction of the object program.

If the OPT S option was selected, the symbol table will be printed in the order the variables were allocated storage by the compiler.

Error Messages: When an error occurs, the source line containing the error is re-printed. On the next line the compiler will print an UP arrow symbol pointing to the position in the source line where either the error was detected OR the last recognized element of the source line before the error occured. This facility is not 100% accurate so the source line should be examined carefully. When an error occurs the processing of the rest of the line is terminated so following statements (if a multiple statement line) are NOT processed.

Listing Format: The first data on a source line is a four-character hex number which is the actual address where the line listed will have its corresponding machine instructions located. The source line is then printed exactly as read.

Loading and Running the Object Program

If no error occured during the compilation process, the object tape may be rewound and loaded using the RT/68 LOAD function just as any other machine language program. The starting address of the program will be $1000 unless an ORG statement was included at the beginning of the program that started it elsewhere.

Debugging the Object Program

Even if the program was compiled correctly it may not execute properly for one of three typical reasons:

1. The program is incorrectly written.
2. Memory was assigned incorrectly - data and program areas may be overlapped (see BASE statement description) or not enough memory is available.
3. Hardware/software relationships (initialization, addresses, operation, etc.) are incorrect, particularly when non-terminal input/output is used.

Very valuable information is contained on the listing and symbol table dump. The RT/68 BREAKPOINT function may be used to place breakpoints at the addresses of each BASIC source line (obtained from the listing) and the breakpoint will occur when the machine language corresponding to that statement is executed. Addresses of variable can be obtained from the symbol table and examined (in hex) to check their value at that point in program execution by using the RT/68 MEMORY EXAMINE/CHANGE or DUMP functions.

Advanced programmers may wish to select the OPT H option to obtain a full instruction listing and place breakpoints at the single instruction level if required.

Memory Assignment

A/BASIC permits total control of memory allocation but two important areas must be preserved:

1) Operating System Area - memory at addresses $0000-$001F are reserved for operating systems and normally should not be used.

2) Run-Time Fast Scratch Storage - 16 bytes of memory at addresses $0020-$002F are used by subroutines in BASLIB for fast working storage. Any programs that use: strings, input/output, or multiplication must not overwrite this area. Memory assignments are:

$20 - XR temp. for strings, I/O
$22 - I/O buffer position pointer
$24 - I/O buffer beg. address
$26 - I/O buffer character count
$27 - string buffer pointer
$29 - string operation XR temp.
$2B - multiplication overflow word, string scratch storage
$2B-$2F - string misc. scratch

Operating System Interfaces

The RT/68 operating system must be present, to compile in all cases, and present for the object program if it uses either input/output to tape or the terminal; or any of the real-time or interrupt capablilities.

If the program is to be run on another system, addresses of equivalent I/O subroutines must be patched into BASLIB before compilation, and no interrupt or real-time statements used in the program.

A number of operating characteristics of the compiler may be changed by "patching" memory locations with different values after the compiler is loaded into memory. A copy of the compiler may then be saved for later use.

Source listings of important I/O regions of the compiler are included for users who have special I/O requirements. These may be modified as desired as long as the subroutine length and addresses are not different. The source listing titled "BAS1.0" is the compile-time I/O system and the listing titled "BASLIB.1" is an "image" of subroutines which may be included in the program generated.

INTERRECORD GAP: The interrecord gap is a delay placed between tape data (or source) records to allow for tape motor start/stop. It is generated by transmitting a series of ASCII NULL (0) characters. This time is equal to the sum of the start and stop times Unless modified, 90 nulls are transmitted which is a 3 second gap at 30 CPS. The addresses of the counts are $17AC for run-time (BASLIB) and $0151 for compile-time. The count may be 0 to 255.

TABLE SIZES: If more than 8K of RAM is available the symbol table and line reference tables may be expanded to permit compilation of longer programs. The table below refers to the constants "X" at addresses $0114-$0115 and "Y" at addresses $011B-$011C which establish the table sizes.

MEMORY SIZE	NUMBER OF SYMBOLS	NUMBER OF LINE REFS.	TYP. PROGRAM LENGTH (LINES)	X	Y
8K	50	100	200-500	1ED2	1FFE
12K	250	824	2000-4000	2A22	2FFE
16K	400	1620	3000-9999	3692	3FF2

CASSETTE PORT CONTROL: A/BASIC as supplied will bring the control PIA (addr $8004-$8007) output CB2 "reader control" to a high state for either tape READ or WRITE to allow RT/68 to automatically switch ports as described in section 8 of the RT/68 Systems Manual. If the cassette interface used does NOT operate properly on RECORD with this signal change addresses $16F6 and $17A7 to $34.

MOTOROLA D2 KIT: The Microware DA-1 or equivalent modifications to the kit must be installed. There are "patch areas" in the I/O subroutines to allow control of the D2 kit's built in cassette interface. The automatic motor control circuit shown in the appendix MUST be used. The following "patches" to the compiler must be made to support D2 cassette operation:

16F4 - 51	17A6 - 51	18DF - 10
1705 - 10	17BE - 10	18FE - 10
1706 - F7	17C4 - B7	1901 - B7
1708 - 80	17C5 - 80	1902 - 80
1709 - 07	17C6 - 08	1903 - 08
170A - B7		
170B - 80		
170C - 08		

RTEDIT

RTEDIT allows the user to enter or modify an A/BASIC source program. The program may use line numbers up to 9999. Each line can contain up to 64 characters. In an 8K system 50 program lines may be entered or edited at a time. For programs longer than 50 lines, repeating the LOAD/SAVE command sequence will read blocks of up to 50 program lines IN/OUT of memory at a time.

COMMANDS

NEW (cr)	Clears memory of all program lines.
RT68 (cr)	Returns user to RT68MX console monitor.
LIST (cr)	Prints all A/BASIC program lines in numeric ascending sequence.
L,<NNNN> (cr)	Lists A/BASIC program line, where <NNNN> is the one to four digit line number of the program line.
L,<BBBB>,<EEEE> (cr)	Lists A/BASIC program lines beginning at line number <BBBB> and ending at line number <EEEE>. <BBBB> and <EEEE> may be one to four digit line numbers and must be seperated by a comma.
MEM (cr)	Directs RTEDIT to print the number of program lines that can be added.
F/<SSS...S> (cr)	The "FIND STRING" function will cause RTEDIT to search the A/BASIC program lines for the first occurance of the character string <S>. String variable <S> may contain numeric, alphabetic, or special characters and can be a maximum of 32 characters. If a match is found the program line containing the match will be printed. If no match is made "NOT FOUND" will be printed.
LOAD (cr)	Causes RTEDIT to read a cassette tape of previously "SAVED" A/BASIC program lines. If the end of file character is read then "*EOF*" is printed.
<NNNN> < A/BASIC > statement	Entering a one to four digit line number followed by a space and an A/BASIC program line will cause the RTEDIT program to search the program lines in memory. If no line number equaling <NNNN> is found the program line will be added. If <NNNN> is found the current program line will replace the previous program line.
<NNNN> (cr)	Deletes program line number <NNNN>.

SAVE (cr)	This function writes all the program lines in memory on cassette tape. A/BASIC program lines will be written to tape in numeric ascending sequence. The tape will be written 128 bytes or 4 lines per record.
S,<BBBB>,<EEEE>	Program lines beginning at line number <BBBB> and ending at line number <EEEE> will be written to cassette tape. Program lines will be written on tape in the same format as "SAVE".
END (cr)	Performs function similar to "SAVE" except that the character "1A" is written as the last character on the cassette tape. Character "1A" signifies end of file (EOF).

NOTES

1. No line longer than 64 characters is allowed.

2. A line number of zero is not permitted.

3. For commands not recognized by RTEDIT the word "WHAT" is printed.

4. Cold start= E,0100 : Warm start= E,013F

5. CONTROL "X" entered before a carriage return will delete the current input line.
   CONTROL "O" entered before a carriage return will delete the last input line, character by character.

6. For users with more than 8K RAM memory the RTEDIT program can be easily modified and recompiled to allow the user to work with up to 127 program lines in memory at a time.

   For 12K system change :
   ```
   010 DIM N(100),B$(200)
   015 V=100 : FOR K=1 TO V : N(K)=0 : NEXT K
   ```

   For 16K system change :
   ```
   010 DIM N(127),B$(254)
   015 V=127 : FOR K=1 TO V : N(K)=0 : NEXT K
   ```

A/BASIC V1.0C ERROR CODES

01 Line number missing
02 Line number duplicated or out of sequence
03 Unrecognized statement
04 Syntax error - specific error not recognizable
05 Variable name missing or in error
06 Equal sign missing
07 Undefined line reference - no such line number
08 Right parenthesis missing/misnested parentheses
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 - too many variable names*
14 Illegal task number
15 Missing, incorrect or wrong type relational operator
16 Delimiter (, or ;) missing
17 Quote missing at end of string
18 Illegal type or missing variable for FOR/NEXT counter
19 Redefined array name
20 Error in array specifcation
21 Size specification zero or greater than 255 in DIM
22 Variable storage overflow - tried to allocate past $FFFF
23 Reference to undeclared array
24 Subscript error in array reference
25 Wrong type or error in function argument
26 Illegal option
27 Unrecognized operator in string expression
28 Unrecognized operator in string expression (+ missing)
29 Arguments in string function in error or wrong type
30 Too many for/next loops active (max is 16)
31 Line reference table overflow*
32 Memory overflow - tried to allocate past $FFFF
33 ON or OFF missing in IRQ statement
34 GOTO or GOSUB missing
35 - up syntax error, nature not identified

* These table overflow errors are not program errors. If more memory is available, either table may be expanded to include a larger number of entries.

```
1000 01 REM PROGRAM : RTEDIT/V1/L1/12-77/TC ------------------------
1000 02 REM COMMANDS : RT,LI,ME,F/,LO,S,,SA,EN,NE,L,,(ADD,REP,DEL)
1000 03 REM COLD START = E,0100 : WARM START = E,013F
1000 04 REM
1000 05 REM INIT/NEW ----------------------------------------------
1000 06 OPT S
1000 07 ORG=$0100
0100 08 BASE=$1300
0100 10 DIM N(50),B$(100)
0100 12 BASE=$30
0100 15 V=50 : FOR K=1 TO V : N(K)=0 : NEXT K
0147 98 REM
0147 99 REM PICK COMMAND ------------------------------------------
0147 100 PRINT : INPUT BUF$ : Z$=LEFT$(BUF$,2)
017A 101 IF Z$<>"RT" THEN 103 : STOP
01A8 103 IF Z$="ME" THEN 900
01D3 105 IF Z$="LI" THEN 200
01FE 107 IF Z$="F/" THEN 800
0229 110 IF Z$="LO" THEN 600
0254 115 IF Z$="S," THEN 310
027F 117 IF Z$="SA" THEN 305
02AA 120 IF Z$="EN" THEN 400
02D5 125 IF Z$="NE" THEN 015
0300 130 IF Z$="L," THEN 500
032B 135 Z=VAL(BUF$) : IF Z>0 THEN 145
0355 140 PRINT "WHAT" : GOTO 100
036E 145 GOSUB 7000 : IF X<=64 THEN 150 : PRINT "LINE TOO LONG"
03A2 146 GOTO 100
03A5 148 REM
03A5 149 REM DELETE/REPLACE/ADD A PROGRAM LINE --------------------
03A5 150 GOSUB 4000 : IF K>V THEN 160
03BA 155 N(K)=0 : IF X>4 THEN 160 : GOTO 100
03E6 160 GOSUB 5000 : IF K<V+1 THEN 175
0407 170 PRINT "MEMORY FULL" : GOTO 100
0427 175 N(K)=Z
0440 180 B$(K)=LEFT$(BUF$,32) : B$(K+V)=MID$(BUF$,33,32)
04B3 185 GOTO 100
04B6 199 REM
04B6 200 REM LIST ALL LINES ---------------------------------------
04B6 210 L=1
04BD 220 GOSUB 2000 : IF T=10000 THEN 100
04D0 230 GOSUB 3000 : GOTO 220
04D6 299 REM
04D6 300 REM SAVE ALL LINES ON TAPE -------------------------------
04D6 305 L=1 : H=9999 : GOTO 320
04E8 308 REM
04E8 309 REM   S,<BBBB>,<EEEE> ROUTINE ----------------------------
04E8 310 GOSUB 6000
04EB 320 A1=0 : GOTO 420
04F4 399 REM
04F4 400 REM END/SAVE ROUTINE -------------------------------------
04F4 410 A1=1 : L=1 : H=9999
050A 420 GOSUB 496
050D 430 GOSUB 2000 : IF L>H+1 THEN 440 : IF T<10000 THEN 470
0540 440 IF A1=0 THEN 450 : BUF$=BUF$+CHR$($1A) : X=X+1
057B 450 IF X=0 THEN 100
058B 460 TWRITE BUF$ : GOTO 100
```

```
0597 470 S1=LEN(B$(S)+B$(S+V))
05D3 480 IF P1+S1+1>128 GOSUB 494
05ED 490 BUF$=BUF$+B$(S)+B$(S+V)+CHR$($D)
0636 492 P1=P1+S1+1 : X=X+1 : IF X>3 GOSUB 494 : GOTO 430
0667 494 TWRITE BUF$
0670 496 P1=0 : X=0 : BUF$="" : RETURN
068B 499 REM
069B 500 REM  L,<BBBB>,<EEEE> ROUTINE ----------------------------------
068B 510 GOSUB 6000
068E 530 GOSUB 2000 : IF T=10000 THEN 100 : IF L>H+1 THEN 100
06C4 540 GOSUB 3000 : GOTO 530
06CA 598 REM
06CA 599 REM  ROUTINE LOADS A PROGRAM TAPE ----------------------------
06CA 600 GOSUB 6500
06CD 630 IF X<4 THEN 100
06DA 640 TREAD BUF$
06E0 650 X=SUBSTR(CHR$($D),BUF$)
0705 655 IF X<>0 THEN 670
0715 660 X=SUBSTR(CHR$($1A),BUF$)
073A 665 IF X=0 THEN 600 : PRINT "*EOF*" : GOTO 100
0764 670 GOSUB 5000 : B$(K+V)="" : N(K)=VAL(BUF$)
07B3 675 IF X<=33 THEN 695
07C5 690 B$(K+V)=MID$(BUF$,33,X-33)
0808 695 B$(K)=LEFT$(BUF$,X-1)
0842 700 X1=LEN(BUF$)-X
085E 710 BUF$=MID$(BUF$,X+1,X1) : GOTO 650
0890 798 REM
0890 799 REM  FIND/<STRING>  ------------------------------------------
0890 800 Z$=MID$(BUF$,3,32)
08B9 810 FOR S=1 TO V : IF N(S)=0 THEN 850
08E1 830 IF SUBSTR(Z$,B$(S)+B$(S+V))<>0 THEN 870
0932 850 NEXT S
094C 860 PRINT Z$+"   NOT FOUND" : GOTO 100
0981 870 GOSUB 3000 : GOTO 100
0987 898 REM
0987 899 REM MEMORY AVAILABLE ROUTINE ---------------------------------
0987 900 GOSUB 6500
098A 910 PRINT X ; "  LINES AVAILABLE"
09B4 920 GOTO 100
09B7 998 REM
09B7 999 REM
09B7 1000 REM SUBROUTINES FOR DECODED COMMANDS ------------------------
09B7 1001 REM
09B7 1998 REM
09B7 1999 REM GET NEXT LINE -------------------------------------------
09B7 2000 J=1 : T=10000
09C6 2020 Q=N(J)
09D7 2050 IF Q=0 THEN 2300
09E7 2100 IF L>Q THEN 2300
09F9 2150 IF Q=L THEN 2400
0A09 2200 IF Q>T THEN 2300
0A1B 2250 T=Q : S=J
0A2B 2300 J=J+1 : IF J<>V+1 THEN 2020
0A58 2350 IF T<>10000 THEN 2450 : RETURN
0A69 2400 T=L : S=J
0A79 2450 L=T+1 : RETURN
0A86 2998 REM
```

```
0A86 2999 REM PRINT LINE ----------------------------------------------
0A86 3000 PRINT B$(S)+B$(S+V) : RETURN
0AC8 3998 REM
0AC8 3999 REM IS LINE # ALREADY IN LINE TABLE ? ----------------------
0AC8 4000 FOR K=1 TO V : IF Z=N(K) THEN 4010 : NEXT K
0B17 4010 RETURN
0B18 4998 REM
0B18 4999 REM IS THERE ROOM IN LINE # TABLE ---------------------------
0B18 5000 FOR K=1 TO V : IF N(K)=0 THEN 5010 : NEXT K
0B5A 5010 RETURN
0B5B 5998 REM
0B5B 5999 REM PICKS LOW + HIGH LINE NUMBER FOR L,/S, -----------------
0B5B 6000 C1=SUBSTR(",",MID$(BUF$,3,5))
0B94 6010 L=VAL(MID$(BUF$,3,4))
0BBC 6020 H=VAL(MID$(BUF$,C1+3,4)) : RETURN
0BEA 6498 REM
0BEA 6499 REM HOW MUCH ROOM IN THE LINE # TABLE ? ---------------------
0BEA 6500 X=0 : FOR K=1 TO V : IF N(K)=0 GOSUB 6505 : NEXT K :RETURN
0C33 6505 X=X+1 : RETURN
0C40 6998 REM
0C40 6999 REM LENGTH OF BUF$ ------------------------------------------
0C40 7000 X=LEN(BUF$) : RETURN
0C59 9998 BASE=$1000
0C59 9999 END

ERRORS  000
VAR LEN $0EA8
PGM LEN $0EDC
PGM END $0FDB

ZZ    1300 00 00
N     1302 32 00
B$    1366 64 00
V     0030 00 00
K     0032 00 00
*     0034 00 00
Z$    0036 00 00
Z     0056 00 00
X     0058 00 00
L     005A 00 00
T     005C 00 00
H     005E 00 00
A1    0060 00 00
S1    0062 00 00
S     0064 00 00
F1    0066 00 00
X1    0068 00 00
*     006A 00 00
J     006C 00 00
Q     006E 00 00
C1    0070 00 00
IO    1000 00 00
ST    1081 00 00
```

SUMMARY OF A/BASIC LANGUAGE

ASSIGNMENT STATEMENTS:

LET	POKE	

CONTROL STATEMENTS:

CALL	IF .. THEN	ON NOVER GOTO
FOR .. TO .. STEP	IF .. GOSUB	ON .. GOTO
NEXT	ON ERROR GOTO	ON .. GOSUB
GOSUB	ON OVR GOTO	STOP
RETURN		

INPUT/OUTPUT:

INPUT	TREAD	TWRITE
PRINT		

SYSTEM CONTROL AND REAL-TIME:

GEN	ON NMI GOTO	SWITCH
IRQ ON	RETI	TASK .. OFF
IRQ OFF	STACK	TASK .. ON
ON IRQ GOTO		

COMPILER DIRECTIVES:

BASE	END	ORG
DIM	OPT	REM

NUMERIC FUNCTIONS:

ABS POS CLK RND PEEK TAB

STRING FUNCTIONS:

ASC	LEN	SUBSTR	VAL
CHR$	LEFT$	MID$	RIGHT$
STR$	TRM$	BUF$	

ARITHMETIC/LOGICAL OPERATORS

+	ADD
-	SUBTRACT
/	DIVIDE
*	MULTIPLY
@	AND
!	OR
%	EXCLUSIVE OR
#	(UNARY) NOT
-	(UNARY) NEGATIVE
+	STRING CONCATENATE

RELATIONAL OPERATORS

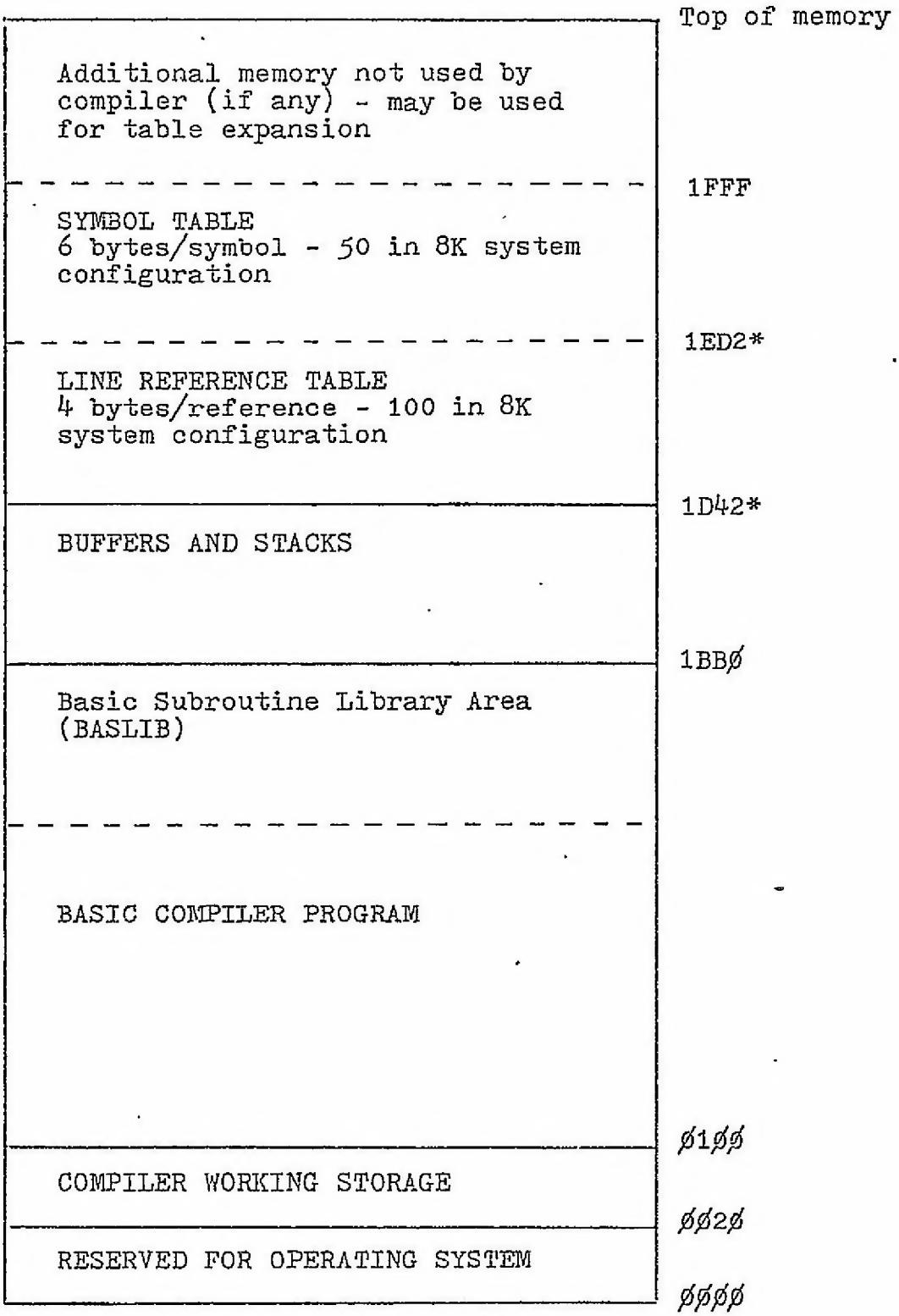
<	LESS THAN
>	GREATER THAN
=	EQUAL
<>	NOT EQUAL
<=	LESS OR EQUAL
=>	GREATER OR EQUAL
<>	STRING NOT EQUAL
=	STRING EQUAL

SPECIAL SYMBOLS:

( )	PARENTHESES
$	HEXADECIMAL PREFIX
:	STATEMENT SEPERATOR
"	STRING DELIMITER

LINE NUMBERS: 1 TO 9999

A/BASIC V 1.0C SYSTEM MEMORY MAP
COMPILE TIME



* These limit addresses are changable by user to expand tables above 8K if desired.

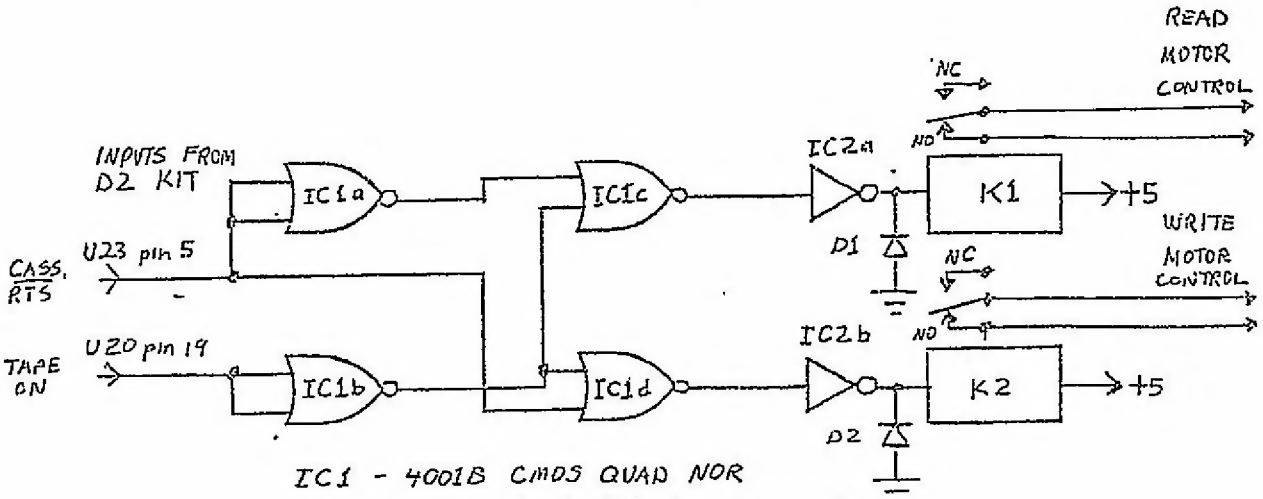
IMPLEMENTATION OF A/BASIC V1.0C ON THE MOTOROLA EVALUATION KIT 2 WITH THE DA-1 UPGRADE KIT

A/BASIC V1.0C has hooks built in to support the MEK6800D2 evaluation kit equipped with the MICROWARE DA-1 upgrade kit.

To support A/BASIC, the D2 kit must be equipped with:

1 - At least 8K bytes of RAM memory starting at address ø.
2 - two cassette tape units and appropriate motor control for each unit.

The compiler must be able to control two tape units, one for reading the source program and one for writing the object tape. Automatic motor control is necessary because the compiler must alternately activate the READ and WRITE cassette units. The schematic below can be used to decode READ ON and WRITE ON from control signals available on the D2 MPU card. These motor control outputs can drive the motor control relay circuit also illustrated. The relay's contacts are connected to a cable and suitable plug (typically a micro phono plug) which is plugged into the cassette recorder's REMOTE jack.



IC1 - 4001B CMOS QUAD NOR
IC2 - 7405 TTL HEX O.C. INVERTER
D1,2 - IN914 DIODE
K1,2 - RELAY, RADIO SHACK 275-004 or EQUIV.

```
03307                   * A/BASIC V1.0C CASSETTE-ORIENTED INPUT/OUTPUT
03308                   * SUBROUTINE PACKAGE

03310                   * JMP TABLE TO RT/68 ROUTINES
03311 16DE 7E E0EA TAPOUT JMP     $E0EA
03312 16E1 7E E3A6 PRNTCH JMP     $E3A6     CONSOLE OUTPUT CHAR
03313 16E4 7E E350 INCH   JMP     $E350     CONSOLE INPUT CHAR
03314      E16A    CLNUP  EQU     $E16A     RT/68 ENTRY POINT
03315 16E7 7E E0C8 OUT4HS JMP     $E0C8     PRINT 4 HEX + SP
03316                   *
03317 16EA 7E E0CA OUT2HS JMP     $E0CA     PRINT 2 HEX + SP
03318 16ED 7E E141 OUTEOL JMP     $E141     PRINT CR+LF
03319 16F0 7E E07E RTPDAT JMP     $E07E     PRINT CHAR. STRING

03321                   * TAPE OUTPUT DEVICE ON
03322 16F3 86 12   OBJFWD LDA A  #$12      TAPE ON CHAR
03323 16F5 C6 3C          LDA B  #$3C      PIA CONTRL BYTE
03324 16F7 8D 0F          BSR    TAPCON
03325 16F9 D6 42          LDA B  NULCNT
03326 16FB 27 06          BEQ    DELAY2
03327 16FD 4F      DELAY  CLR A
03328 16FE 8D E1          BSR    PRNTCH
03329 1700 5A             DEC B
03330 1701 26 FA          BNE    DELAY
03331 1703 39      DELAY2 RTS

03333                   * TAPE OUTPUT DEVICE OFF
03334 1704 86 14   OBJSTP LDA A  #$14      TAPE OFF CHAR
03335 1706 C6 34          LDA B  #$34      PIA CNTRL BYTE

03337 1708 8D D7   TAPCON BSR    PRNTCH
03338 170A F7 8007        STA B  $8007
03339 170D 01             NOP
03340 170E 39             RTS

03343                   * SUBR TO REWIND SOURCE FILE
03344 170F 8D DC   SRCREW BSR    OUTEOL
03345 1711 CE 175A        LDX    #REWMSG
03346 1714 8D DA          BSR    RTPDAT    PRINT MESSAGE
03347 1716 20 CC          BRA    INCH      WAIT FOR CHR

03349                   * SUBR TO GET SOURCE LINE
03350 1718 C6 48   NEWBUF LDA B  #72
03351 171A 8D 27          BSR    NEWBF9
03352 171C 37      NEWBF2 PSH B             SET MAX CHAR COUNT
03353 171D DE F3   NEWBF3 LDX    SRCPTR    GET TAPE BUF PTR
03354 171F A6 00          LDA A  0,X       GET NEXT CHR
03355 1721 26 04          BNE    NEWBF4    BRA IF NOT END OF RECORD
03356 1723 8D 25          BSR    SRCREC    READ A NEW TAPE RECORD
03357 1725 20 F6          BRA    NEWBF3
```

```
03358 1727 08       NEWBF4 INX              ADV TAPE BUF PTR
03359 1728 DF F3           STX     SRCPTR   SAVE IT
03360 172A 33              PUL B
03361 172B 81 1A           CMP A   #$1A
03362 172D 26 02           BNE     NEWBF5
03363 172F 0C              CLC
03364 1730 39              RTS
03365 1731 DE 31    NEWBF5 LDX     BUFPTR   GET LINE BUF PTR
03366 1733 A7 00           STA A   0,X      DROP IN THE CHR
03367 1735 08              INX
03368 1736 DF 31           STX     BUFPTR
03369 1738 81 0D           CMP A   #$D
03370 173A 27 07           BEQ     NEWBF9
03371 173C 5A              DEC B
03372 173D 26 DD           BNE     NEWBF2
03373 173F 86 0D           LDA A   #$D
03374 1741 A7 00           STA A   0,X
03375 1743 CE 1CF9 NEWBF9  LDX     #LINBUF
03376 1746 DF 31           STX     BUFPTR
03377 1748 0D              SEC
03378 1749 39              RTS

03380                      * SUBR TO READ A TAPE RECORD, USES SUBR
03381                      * "ZRDBUF" FROM BASLIB
03382 174A 8D 03    SRCREC BSR     SREC2    INIT PTRS
03383 174C BD 18DE         JSR     $18DE    CALL SUBR
03384 174F CE 1C78 SREC2   LDX     #TAPBUF
03385 1752 DF F3           STX     SRCPTR
03386 1754 39              RTS

03388                      * SUBR TO INIT I/O
03389 1755 8D F8    INZIO  BSR     SREC2
03390 1757 6F 00           CLR     0,X
03391 1759 39              RTS
03392 175A 50       REWMSG FCC     /PASS2/
03393 175F 04              FCB     $4
```

```
00010                           NAM     BASLIB.1
00020                           OPT     O
00030                   * BASLIB V1.1 BASIC SUBROUTINE LIBRARY FOR
00040                   * A/BASIC V1.0C - CASSETTE ORIENTED
00050                   * 11/2/77 K. KAPLAN

00070                   * RUN TIME STORAGE
00080 0020                      ORG     $20
00090 0020 0002         XSAVE   RMB     2
00100 0022 0002         BUFPTR  RMB     2
00110 0024 0002         BUFBEG  RMB     2
00120 0026 0001         ZONE    RMB     1
00130 0027 0002         STRPTR  RMB     2
00140 0029 0002         STRSAV  RMB     2

00160                   * RT/68 REFERENCES
00170       E141        CRLF    EQU     $E141
00180       E0CC        OUTSP   EQU     $E0CC
00190       E3A6        OUTCH   EQU     $E3A6

00210       E350        INCH    EQU     $E350
00220       E07E        PDATA   EQU     $E07E

00240       8007        PIACB   EQU     $8007

00260 1760                      ORG     $1760
00270                   *PUT SPACE IN I/O BUFFER
00280 1760 86 20        ZOUTS   LDA A   #$20

00300                   *PUT CHAR IN I/O BUFFER
00310 1762 DF 20        ZOUT    STX     XSAVE
00320 1764 DE 22                LDX     BUFPTR
00330 1766 A7 00                STA A   0,X
00340 1768 96 26                LDA A   ZONE
00350 176A 81 80                CMP A   #128        BUFFER LIMIT
00360 176C 22 02                BHI     ZOUT2
00370 176E 4C                   INC A
00380 176F 08                   INX
00390 1770 6F 00        ZOUT2   CLR     0,X         MARK END OF BUFFER
00400 1772 97 26                STA A   ZONE
00410 1774 DF 22                STX     BUFPTR
00420 1776 DE 20                LDX     XSAVE
00430 1778 39                   RTS

00450                   *INITIALIZE I/O
00460 1779 DF 24        RESET   STX     BUFBEG
00470 177B 20 04                BRA     ZINZ2
00480 177D 86 01        ZINZIO  LDA A   #1
00490 177F 97 26                STA A   ZONE
00500 1781 DE 24        ZINZ2   LDX     BUFBEG
```

```
00510 1783 DF 22          STX     BUFPTR
00520 1785 39             RTS

00540                *SKIP ZONE
00550 1786 8D D8     ZSKPZ2 BSR    ZOUTS
00560 1788 96 26     ZSKPZ  LDA A  ZONE
00570 178A 84 07            AND A  #7
00580 178C 81 01            CMP A  #1
00590 178E 26.F6            BNE    ZSKPZ2
00600 1790 39               RTS

00620                * PRINT BUFFER - NO CR/LF
00630 1791 8D EE     ZENDBN BSR    ZINZ2
00640 1793 A6 00     ZEBN2  LDA A  0,X
00650 1795 27 0B            BEQ    ZTAB3
00660 1797 8D 2F            BSR    PRCHR
00670 1799 08               INX
00680 179A 20 F7            BRA    ZEBN2

00700 179C 8D C2     ZTAB2  BSR    ZOUTS
00710 179E D1 26     ZTAB   CMP B  ZONE
00720 17A0 22 FA            BHI    ZTAB2
00730                * TAB FUNCTION
00740 17A2 39        ZTAB3  RTS

00760 17A3 20 BD     ZGOUT  BRA    ZOUT

00780                * WRITE BUFFER TO TAPE
00790 17A5 86 12     ZENDBT LDA A  #$12      TAPE ON CODE
00800 17A7 C6 3C            LDA B  #$3C
00810 17A9 8D 16            BSR    TAPCON
00820 17AB C6 5A            LDA B  #90
00830 17AD 4F        ZENDT2 CLR A
00840 17AE 8D 18            BSR    PRCHR
00850 17B0 5A               DEC B
00860 17B1 26 FA            BNE    ZENDT2
00870 17B3 86 02            LDA A  #2
00880 17B5 8D 11            BSR    PRCHR
00890 17B7 8D D8            BSR    ZENDBN
00900 17B9 86 03            LDA A  #3
00910 17BB 8D 0B            BSR    PRCHR
00920 17BD 86 14            LDA A  #$14
00930 17BF C6 34            LDA B  #$34
00940 17C1 F7 8007 TAPCON   STA B  PIACB
00950 17C4 8D 02            BSR    PRCHR
00960 17C6 01               NOP             PATCH AREA FOR D2 KIT
00970 17C7 39               RTS
00980 17C8 7E E3A6 PRCHR    JMP    OUTCH
```

```
01000                  * PRINT BUFFER + CR/LF
01010 17CB 8D C4       ZENDB  BSR     ZENDBN
01020 17CD 8D AE       ZCRLF  BSR     ZINZ10
01030 17CF 7E E141            JMP     CRLF
```



```
02870                  * READ TAPE DATA RECORD
02880 18DE 86 11       ZRDREC LDA A   #$11
02890 18E0 C6 3C              LDA B   #$3C      CONTROL CODES
02900 18E2 8D 1D              BSR     XCNTRL    TURN ON LOAD
02910 18E4 BD E350     ZREC2  JSR     INCH      SEARCH FOR SOR
02920 18E7 81 02              CMP A   #2        START OF RECORD CHAR?
02930 18E9 26 F9              BNE     ZREC2
02940 18EB 5F                 CLR B             SET UP CHAR. COUNT
02950 18EC BD E350     ZREC3  JSR     INCH
02960 18EF 81 03              CMP A   #3
02970 18F1 27 08              BEQ     ZREC4     EOR DETECTED
02980 18F3 A7 00              STA A   0,X
02990 18F5 08                 INX
03000 18F6 5C                 INC B             BUMP CHAR COUNT
03010 18F7 C1 80              CMP B   #128      BUFFER FULL?
03020 18F9 25 F1              BCS     ZREC3     GET ANOTHER IF NOT
03030 18FB 6F 00       ZREC4  CLR     0,X       MARK END OF BUFFER
03040 18FD 86 13              LDA A   #$13      READ OFF CODE
03050 18FF C6 34              LDA B   #$34      PIA CNTRL BYTE
03060 1901 BD E3A6     XCNTRL JSR     OUTCH     XMIT CONTROL CODE
03070 1904 F7 8007            STA B   PIACB
03080 1907 39                 RTS
```

```
03100                  * READ TERMINAL INPUT LINE
03110 1908 86 3F   ZRDLIN LDA A  #'?
03120 190A BD E3A6        JSR    OUTCH
03130 190D BD E0CC        JSR    OUTSP
03140                  * READ INPUT BUFFER
03150 1910 DF 22   ZRDBUF STX    BUFPTR
03160 1912 DF 24          STX    BUFBEG
03170 1914 C6 80          LDA B  #128
03180 1916 BD E350 ZRBUF2 JSR    INCH
03190 1919 81 0F          CMP A  #$F        BACKSPACE?
03200 191B 26 0C          BNE    CHKDEL
03210 191D 9C 24          CPX    BUFBEG
03220 191F 27 0C          BEQ    LINKIL
03230 1921 09             DEX
03240 1922 A6 00          LDA A  0,X
03250 1924 BD E3A6        JSR    OUTCH
03260 1927 20 ED          BRA    ZRBUF2
03270 1929 81 18   CHKDEL CMP A  #$18
03280 192B 26 17          BNE    ZRBUF4
03290                  * DELETE LINE
03300 192D 8D 07   LINKIL BSR    ZRBUF3
03310 192F 20             FCC    / *DEL*/
      1930 2A
      1931 44
      1932 45
      1933 4C
      1934 2A
03320 1935 04             FCB    4
03330 1936 30      ZRBUF3 TSX
03340 1937 EE 00          LDX    0,X
03350 1939 31             INS
03360 193A 31             INS
03370 193B BD E07E        JSR    PDATA
03380 193E 8D 0C          BSR    ZRLIN3
03390 1940 DE 24          LDX    BUFBEG
03400 1942 20 CC          BRA    ZRDBUF
03410 1944 A7 00   ZRBUF4 STA A  0,X
03420 1946 81 0D          CMP A  #$D
03430 1948 26 09          BNE    ZRBUF5
03440 194A 6F 00          CLR    0,X
03450 194C 86 01   ZRLIN3 LDA A  #1
03460 194E 97 26          STA A  ZONE
03470 1950 7E E141        JMP    CRLF
03480 1953 5D      ZRBUF5 TST B
03490 1954 27 C0          BEQ    ZRBUF2
03500 1956 08             INX
03510 1957 5A             DEC B
03520 1958 20 BC          BRA    ZRBUF2
```