MICROSOFT

BASIC COMPILER

SORCERER

BASIC

COMPILER

PREFACE

SYSTEM SOFTWARE is one of the leading international suppliers of software for the Sorcerer Computer. SYSTEM SOFTWARE has successfully researched, developed and marketed more than 20 top quality software products for Sorcerer computers to more than 30 countries. Besides its direct mailing list of over 2000 clients, System Software distributes software through software houses and dealers throughout the world.

The BASIC COMPILER is the latest quality product from System Software. It is the result of over 3500 man hours of research, development and testing by a team of highly skilled programmers and language designers.

The design objectives of the BASIC COMPILER were to produce a fast, compact and user friendly system to meet a wide range of user requirements. These objectives have been met and the BASIC COMPILER system, while maintaining compatability with the Sorcerer ROM PAC system, includes a number of additional powerful features. These features include integer variables, integer and byte arrays, graphics commands, cassette string input/output, cursor control, a special keyboard input command, extra string handling facilities and more.

The compiler can be used for home and family use, for business applications and scientific research. Often a compiled program will eliminate the need to write routines in Assembly Language because of the speed advantage.

This BASIC Compiler is probably the largest, the most complex and worthwhile project ever undertaken and written exclusively for the Sorcerer Computer.

SYSTEM SOFTWARE is committed to continuing research and development of new and better software products for Sorcerer users. Developments in micro-computer software, and users' needs are continually monitored to ensure that high performance products are produced to meet the users' needs.

WARRANTY & BACKUP

SYSTEM SOFTWARE has made every effort to ensure that the BASIC COMPILER operates as specified. However, no claims or warranties are made with respect to the BASIC COMPILER or any associated materials. In particular, no claims are made as to the fitness of the BASIC COMPILER for any particular purpose.

From time to time upgrades, having extra commands and facilities will become available. These improvements will be available to original purchasers of the BASIC COMPILER at a reduced price.

If you find what you consider a problem with either the compiler or documentation, please inform SYSTEM SOFTWARE. If a bug is detected, please mention the version number, any modification you may have had done to your computer or ROM PAC, the amount of RAM you are using and a listing of the problem with an explanation of the symptoms.

COPYRIGHT

CONTENTS

1. INTRODUCTION TO THE BASIC COMPILER
   1.1 LOADING THE COMPILER FROM CASSETTE
2. BASIC LANGUAGE
   2.1 PROGRAM STRUCTURE
      2.1.1 Statement Order
   2.2 LANGUAGE ELEMENTS
      2.2.1 Notation
      2.2.2 Character Set
      2.2.3 Line Format
      2.2.4 Constants
      2.2.5 Variable Names
      2.2.6 Scalar Variables
      2.2.7 Array Variables
      2.2.8 Function Values
      2.2.9 Operators
      2.2.10 Space Requirements for Variables and Constants
   2.3 EXPRESSIONS
      2.3.1 Arithmetic Operators
      2.3.2 Numeric Overflow
      2.3.3 String Concatenation
      2.3.4 Relational Operators
      2.3.5 Logical Operators
      2.3.6 Order of Expression Evaluation
      2.3.7 Type Conversion
   2.4 SPECIFICATION STATEMENTS
      2.4.1 **REM\OPTION**
      2.4.2 **DIM**
      2.4.3 **REM\BYTE**
      2.4.4 **REM\INTEGER**
      2.4.5 **CLEAR**
      2.4.6 **REM**
   2.5 ASSIGNMENT STATEMENTS
      2.5.1 **LET**
      2.5.2 **MID$**
   2.6 FLOW CONTROL STATEMENTS
      2.6.1 **GOTO**
      2.6.2 **GOSUB**
      2.6.3 **RETURN**
      2.6.4 **IF - THEN - ELSE**
      2.6.5 **FOR - NEXT**
      2.6.6 **ON**
      2.6.7 **STOP**
      2.6.8 **END**

2.7 INPUT/OUTPUT STATEMENTS
2.7.1 **INPUT**
2.7.2 **READ**
2.7.3 **DATA**
2.7.4 **RESTORE**
2.7.5 **PRINT** and **PRINT&**
2.7.6 **SET**
2.7.7 **RESET**
2.7.8 **CLOAD***
2.7.9 **CSAVE***
2.7.10 **POKE**
2.7.11 **OUT**
2.7.12 **WAIT**
2.8 USER DEFINED FUNCTIONS
2.8.1 **DEF FN**
2.9 USER ASSEMBLY ROUTINES
2.9.1 **USR**
2.10 BASIC FUNCTIONS
2.10.1 Numeric - **ABS,EXP,INT,LOG,SGN,SQR,RND**
2.10.2 Trigonometric - **ATN,COS,SIN,TAN**
2.10.3 String - **ASC,CHR$,CVI,CVS,INSTR,LEFT$,LEN MID$,MKI$,MKS$,RIGHT$,SPC,STR$,VAL**
2.10.4 Input/Output - **INKEY,INP,PEEK,POS,SPC,TAB**
2.10.5 General - **FRE**

3. PROGRAM DEVELOPMENT
3.1 DIFFERENCES BETWEEN ROM PAC BASIC AND BASIC COMPILER
3.2 OPTIMIZING PROGRAM PERFORMANCE
3.3 APPLICATION EXAMPLES
3.3.1 Printer Interface
3.3.2 Using the Compiler with Disks
3.3.3 Full Example in Using the Compiler

4. COMPILER OPERATING INSTRUCTIONS

APPENDICES
Appendix A - Error Messages
Appendix B - Reserved Words
Appendix C - Memory Maps

CHAPTER 1. INTRODUCTION TO THE BASIC COMPILER

The compiler is designed to run in a 32K or 48K Exidy Sorcerer computer system with one or more cassette drives. It is simple to operate, allowing the user to develop or edit his BASIC program with the standard BASIC ROM PAC editor and then compile the resulting program directly from the source program in memory.

The BASIC compiler is a program which converts a source program written in BASIC into object code, which may then be run at a speed comparable to machine code. The code produced is the machine code for a hypothetical stack orientated computer which is emulated in the Sorcerer by a machine code Runtime Support System. In general, the code produced by the compiler is more compact than the original tokenised code. At the time of compilation, the compiler checks the syntax of the source program, resolves all line numbers referenced, translates numeric constants to binary and fixes the locations of all variables used in the original BASIC program. An interpreter, on the other hand, is at a speed disadvantage because it must carry out the same functions every time it executes each statement. Compilation is carried out only once, and the resulting object code may then be stored on cassette or disk for subsequent use, or immediately executed.

1.1 LOADING THE COMPILER FROM CASSETTE

The supplied cassette tape contains a 32K version of the compiler on the front side and a 48K version on the reverse side. Both recordings are at 300 baud. It is suggested that the user makes a copy of the compiler on his own cassette at 1200 baud and keep the supplied cassette as a backup copy.

To copy the 32K version:

```
SE T=1
LO
SE T=0
SE X=4C6E
SA COM32 2800 7EFF
```

To copy the 48K version:

```
SE T=1
LO
SE T=0
SE X=8B6E
SA COM48 6700 BDFF
```

To load and execute the compiler, turn on the Sorcerer with the BASIC ROM PAC inserted, (you may then optionally write or load a BASIC program) and type: BYE followed by LOG. Load the appropriate version of the compiler. Upon loading, the compiler will try to compile the program in memory. If none is found, a ON ERROR will be printed and control will be returned to BASIC. If a Basic Source program is present, it will be compiled. To execute the compiler at any time from BASIC, type: BYE followed by GO 100

CHAPTER 2. BASIC LANGUAGE

This chapter describes the syntax and purpose of each BASIC statement processed by the BASIC COMPILER. The BASIC COMPILER includes all the BASIC program statements in the ROM PAC BASIC and in addition supports extra statements and functions which can be used to increase speed and reduce space requirements.

In general, any program which runs with ROM PAC BASIC will also run with the BASIC COMPILER. Refer to chapter 3, 'Program Development' for the differences between ROM PAC BASIC and the BASIC COMPILER, and for useful hints on optimizing program performance.

2.1 PROGRAM STRUCTURE

A BASIC program consists of one or more lines of text. Each line begins with a line number which is followed by one or more BASIC statements separated by colons. (:) A BASIC statement consists of a keyword, (or an implied LET), followed by various language elements such as constants, variables, operators and functions.

There are five main types of BASIC statements:

1) REM statements - to make the program more readable.
2) Specification statements - to specify what types the variables are.
3) Assignment statements - to evaluate expressions and store the result.
4) Flow Control statements - to change the order of program execution.
5) Input/Output statements - to transfer data.

2.1.1 STATEMENT ORDER

The BASIC COMPILER requires that certain statements (if present) must appear at the beginning of a BASIC program, in the following order:

```
1) OPTION or REM\OPTION
2) DIM
3) REM\BYTE
4) REM\INTEGER
5) Other statements
```

These statements are described in section 2.4 'Specification Statements'. REM statements (other than the above REM\ statements) may appear anywhere in a program, including before and interleaved with the above statements.

2.2 LANGUAGE ELEMENTS

This section describes the language elements which are used in BASIC statements. The rules for combining language elements in expressions are presented in section 2.3 and the various BASIC statements themselves are described in section 2.4 to 2.7

2.2.1 NOTATION

The following terminology is used to describe the syntax of BASIC statements and BASIC functions.

```
[ ]             The contents are optional
UPPER CASE      BASIC reserved words
lower case      Words or symbols supplied by the user
...             Indicates that the portion of the statement
                immediately preceding the dots may be repeated
I,J             represent integer expressions
X,Y             represent real expressions
X$,Y$           represent string expressions
```

2.2.2 CHARACTER SET

The set of characters recognised by the BASIC Compiler consist of:

1) Digits 0-9
2) Upper case and lower case letters of the alphabet.(A-Z,a-z)
3) The following operators and delimiters: + - * \ / =
   < > ( ) " . $; : ^ ,SPACE
4) The carriage return <CR> terminates each input line

In addition, CTRL C and RUNSTOP are used by the Run Time Support System to abort and interrupt execution respectively.

2.2.3 LINE FORMAT

Each line of source code takes the form:

```
line number  BASIC statement [:BASIC statement]...  <CR>
```

The line number is a whole number between 0 and 65535 inclusive and a line may have up to 64 characters including the line number.

2.2.4 CONSTANTS

Constants are language elements which have a fixed value in a program. They may be logical, byte, integer, real or string.

LOGICAL CONSTANTS

Only two values are recognised. False is represented by the value zero, while true is any nonzero numeric value. When a logical expression is evaluated, the result will be set to 0 or -1

BYTE CONSTANTS

Whole numbers between 0 and 255 inclusive that are not followed by a decimal point. eg. 0, 47, 255

INTEGER CONSTANTS

Whole numbers between -32768 and 32767 inclusive and must be written without a decimal point. eg. 0, 47, 255, 280, 32767, -50,

REAL CONSTANTS

Positive and negative numbers containing a decimal point or number expressed in exponential form. In this latter form, a number (the mantissa) is followed by the letter E and a signed or unsigned integer (the exponent). The exponent represents the power of 10 by which the mantissa is multiplied. In memory, a real number is contained in 4 bytes. The range of the absolute value of a real constant is between about 1.4E-38 and 1.4e+38 with 6 significant digits.
eg. 0., -32., 5.007, 1.3E-32

STRING CONSTANTS

Consist of zero to 255 alphanumeric characters delimited by double quotation marks. Any character what-so-ever may be included in a string except a double quote. eg. "ABC123", "Hello"

The closing quote is optional if the string is the last entry on a line. A null string is expressed as two adjacent quote marks. ie ""

2.2.5 VARIABLE NAMES

A variable is a language element whose value can be changed during program execution. The two main types of variables are scalar variables (which store a single value) and array variables. (specified with a DIM statement and can store more than one value).

A variable name begins with a letter followed by any number of letters or digits followed by an optional '$'. Only the first 2 characters (excluding the '$') are significant. The names used must not be BASIC Compiler reserved words.
eg. AB, AB$, A12 are legal variable names.

Variables are assumed to be numeric, unless the name is followed by $, in which case it is assumed to be a string variable. The name before the dollar sign may be the same as the name of a numeric variable. ie. AB$ and AB are separate variables and may appear in the same program

If the first 2 characters of a name are FN, the name is that of a user defined function. (see section 2.8)

2.2.6 SCALAR VARIABLES

A scalar variable stores a single value. There are 3 types of scalar variables:

INTEGER SCALAR VARIABLES

An integer scalar variable is specified in a REM\INTEGER statement and its name cannot end in a '$'. An integer scalar variable occupies two bytes of memory and can store a value in the range -32768 to 32767.

```
eg. REM\INTEGER A,I,J
    A=32:I=-4678:I=A+J
```

REAL SCALAR VARIABLES

A real scalar is any scalar variable which neither ends with a '$' nor is specified in a REM\INTEGER statement. A real variable occupies four (4) bytes of memory and can store a value about in the range 1.4E-38 to 1.4E+38. eg. A=52.76 : B5=-1.3006E+23 : X=Y+15.2

STRING SCALAR VARIABLES

The name of a string scalar variable ends in a '$'. A string scalar occupies 4 bytes in memory in addition to the 0 to 255 bytes (characters) stored in the variable itself.
eg. A$="":ABC$="HELLO":X$=Y$+"ABC"

Note: Integer scalar variables should be used wherever possible (in preference to real or string scalar variables) in order to reduce program space and increase execution speed. (See section 3.2)

2.2.7 ARRAY VARIABLES

An array variable stores one or more values. All array variables must be dimensioned in a DIM statement (See section 2.4.2). An array variable cannot have the same name as a scalar variable. The base address of array subscripts is 0.

```
DIM A(5),B(6),I5(2,7)
DIM AJ$(2,4,3)
```
Thus array A has 6 elements: A(0),A(1),...A(5)

There are four types of array variables:

BYTE ARRAY VARIABLES

A byte array variable is specified in a DIM statement and in a REM\BYTE (see section 2.4.3). A byte array name cannot end in a '$'. Each element of a byte array occupies one byte of memory and can store a value in the range 0 to 255.

```
eg. DIM A(5),I2(7,2,19),C(15),J(19,2,4)
    REM\BYTE I2,J
```
(Array I2 and J are byte array variables)

INTEGER ARRAY VARIABLES

An integer array variable is specified in a DIM statement and in a REM\INTEGER statement in a similar way to the specification of Byte Array Variables above. Each element in an integer array occupies 2 bytes of memory and can store a value in the range -32768 to 32767.

REAL ARRAY VARIABLES

A real array variable is specified in a DIM statement. The name cannot end in '$' and it must not be specified in a REM\ statement. Each element occupies 4 bytes of memory and can store a value in the range of about 1.4E-38 to 1.4E+38.

STRING ARRAY VARIABLES

A string array variable is specified in a DIM statement and its name ends in a '$'. Each element of a string array occupies 4 bytes of memory in addition to the 0 to 255 bytes (characters) stored in the element itself.

Byte or integer arrays should be used wherever possible (in preference to real or string arrays) in order to reduce program space and increase execution speed (see section 3.2).

2.2.8 FUNCTION VALUES

Functions are used in expressions to provide a single value. A function value consists of a function name followed by '(', one or more expressions separated by ',', and ending with a ')'. The values of the above expressions are used to calculate the value of the function.

There are two kinds of functions:

USER DEFINED FUNCTIONS

These functions are defined by the user in his program (see section 2.8), and can only return a real value.

```
eg. DEF FNAB(X)=1+2*X
    Y=2+FNAB(3)
```

The value of FNAB(3) is 7.

BASIC FUNCTIONS

The BASIC COMPILER includes a number of pre-defined functions (see section 2.10). BASIC functions return integer, real and string values.

```
eg. Y=SIN(3)+ASC("A")+VAL(LEFT$(STR$(B)))
```

2.2.9 OPERATORS

Operators are used to combine constants, variables and function values into expressions. The operators available are:

```
Arithmetic: ^, -, *, \, /, +
String Concatenation: +
Relational:   =, <>, <, >, <=, >=
Logical    :   NOT, AND, OR, XOR
```

2.2.10 SPACE REQUIREMENTS FOR VARIABLES AND CONSTANTS

The memory requirements for constants and variables are set out as follows:

Constants	Bytes of memory used
Byte	1
Integer	2
Real	4
String	4 + length of the string
Scalar Variables	
Integer	2
Real	4
String	4 + length of string
Array Variables	10 + bytes for each array element as set out for scalar variables above.

2.3 EXPRESSIONS

An expression consists of one or more operands (constants, variables or function references), which are combined with operators to calculate a single value. The rules for forming and evaluating expressions using the different kinds of operators are described in the following sub-sections.

2.3.1 ARITHMETIC OPERATORS

^	Exponentiation	A^B
-	Negation	-A
*	Multiplication	A*B
/	Real division (with result rounded)	A/B
\	Integer division (with fractional part truncated)	A\B
+	Addition	A+B
-	Subtraction	A-B

These operators are described in more detail below.

^ EXPONENTIATION

The value of the expressions A^B is A multiplied by itself B times. A and B may be integer or real values. If either is an integer value, it is firstly converted to a real before exponentiation is carried out.

+ UNARY PLUS AND MINUS OPERATORS

The unary plus and minus operators (together with the logical NOT operator) are the only operators which operate on a single operand. They can appear at the beginning of expressions or immediately after a "(" or at the beginning of a subscript expression. eg. -5.2, A(-3+I), +X

* MULTIPLICATION OPERATOR

The value of A*B is A multiplied by B. A and B can be integer or real values. If A and B are not the same type, the integer value is converted to a real before the multiplication is done.

/ REAL DIVISION OPERATOR

The value of A/B is A divided by B. A and B can be integer or real values. If either A or B is an integer value then it is converted to a real before the two values are divided.

\ INTEGER DIVISION OPERATOR

The value of A\B is A divided by B with the result truncated to an integer. A and B can be integer or real values. If either A or B is a real, the value is firstly converted to an integer before the division is carried out.

EXPRESSIONS	PRELIMINARY VALUE	FINAL VALUE
4\2	2	2
4\3	1.3	1
4\5	0.8	0
-4\3	-1.33	-1
5.3\1.6	3.3125	3

+, - ADDITION AND SUBTRACTION OPERATORS

The value of A+B is the sum of A and B. The value of A-B is the difference between A and B. A and B may be integer or real. If either A or B is real, both values are converted to real before the operation is carried out.

2.3.2 NUMERIC OVERFLOW

Numeric overflow occurs during the evaluation of an expression if the current value of the expression is too large to be stored for the type of expression value. An error message is generated and execution stops.

Overflow occurs in integer expressions if the current value of the expression is outside the range -32768 to +32767. Overflow occurs in real expressions if the current value of the expression is greater than about 1.4E+38.

The order in which an expression is evaluated can determine if overflow will occur as shown in the following examples of integer arithmetic.

EXPRESSION	INTERMEDIATE VALUES	
-32767-4+5	-32771	Overflow occurs
-32767+5-4	-32762, -32766	No overflow

2.3.3 STRING CONCATENATION OPERATOR

The string concatenation operator ('+') is used to concatenate (join) two string values. For example if A$="ABC" and B$="XYZ" then A$+B$ has the value "ABCXYZ".

If two null strings are concatenated the result is a null string. If the combined lengths of the two strings exceeds 255 characters an error message will be printed and execution stopped.

2.3.4 RELATIONAL OPERATORS

Relational operators are used to compare two values. The result of the comparison is an integer value which is either true (-1) or false (0). Relational operators have a lower precedence than arithmetic (or string concatenation) operators.

Real and integer values can be compared with each other. If one of the values is real then the other value is converted to a real before the comparison is made. A string value can be compared with another string value but not with a numeric value.

=	equality	A=B or A$=B$
<> or ><	inequality	A<>B or A$<>B$
<	less than	A<B or A$<B$
>	greater than	A>B or A$>B$
<= or =<	less than or equal	A<=B or A$<=B$
>= or =>	greater than or equal	A>=B or A$>=B$

Strings are compared by comparing the ASCII codes of the characters from left to right until the codes differ or one or both strings are exhausted. The string with the lower code precedes the string with the higher codes. If one string is exhausted before a difference is detected, that string precedes the longer string.

ie.

```
A$=B$ if A$="ABC" and B$="ABC"
A$<B$ if A$="ABC" and B$="ABCD"
A$<B$ if A$="" and B$="ABC"
```

2.3.5 LOGICAL OPERATORS

Logical operators operate on two logical values to produce another logical value. A logical value is an integer which has a value of 0 (false) or non zero (true). If either or both of the values operated on are real then they are rounded up to integers before performing the operation. If either of the values is a string value an error message will be printed and execution stopped. Logical operators have a lower precedence than relational and arithmetic and string concatination.

The logical operators (in order of precedence) and truth table examples are shown below:

I	J	NOT J	I AND J	I OR J	I XOR J
0	0	1	0	0	0
0	1	0	0	1	1
1	0	1	0	1	1
1	1	0	1	1	0

Logical operators are used in compound IF statements and in expressions where it is required to examine or set individual bits in an 8 bit byte or in a 16 bit integer. Integer values are stored in two's complement form.

```
Example: 10 IF A<B AND C<D OR A$>B$ THEN 100
         20 K=6 AND A OR I
```


2.3.6 ORDER OF EXPRESSION EVALUATION

The order of evaluation of expressions is from left to right if operators have the same level of precedence. Otherwise, the operators with the highest order of precedence are evaluated first. The order of precedence (which may be changed by the use of parentheses) is given below.

Order	Operation
1.	Expressions within parentheses
2.	^
3.	- (negation)
4.	*, \, /
5.	+, -
6.	=, <>,<, >, <=, >=
7.	NOT
8.	AND
9.	OR, XOR

2.3.7 TYPE CONVERSION

Expressions will be evaluated according to the rules of precedence outlined in section 3.7. Byte and Integer values are converted to real when they are combined with a real value by an operator.

```
eg. 3*10. will result in the floating of the byte constant 3
    before it is multiplied by the real constant 10. to give
    the real result 30. (Note that real values contain
    decimal points)
```

Floating point division, /, always results in the conversion of both operands to real numbers prior to carrying out the division.

```
eg. 3/30 will give the real result 0.3
    13.\10. will give the integer result 1
```

Type conversion from integer to real or vice versa will be carried out automatically when results are used as subscripts or function arguments or when the results of expressions are stored in numeric variables. When an integer result is required, a real value will be rounded and when a real result is required the integer value will be floated.

2.4 SPECIFICATION STATEMENTS

Specification statements are used to specify compiler options and to specify variable types. Specification statements (if present) must appear at the beginning of a program and in the same order as below, except that the REM statement may appear anywhere in a program.

2.4.1 REM\OPTION

Syntax: REM\OPTION 1 or 0

Purpose: Specifies whether the line number of each program line is to be output in the compiled code. If an error occurs during execution of the compiled program the line number of the statement in error will be printed in the error message

Note 1) If the statement is omitted or REM\OPTION 0 is specified then line numbers will not be output. If OPTION 1 is specified, line numbers will be output.
2) The statement is useful when debugging a program, however each line number output takes 3 bytes of memory and increases execution time.
3) The statement must appear on a line by itself. REM\OPTION will, of course, be ignored by ROM PAC BASIC.

2.4.2 DIM

Syntax: DIM list of subscripted array variables

Purpose: Specifies the maximum subscript values for each array.

Note 1) A maximum of 3 subscripts are allowed.
2) The base value for each subscript is 0 and the maximum is 32767.
3) No array element may be referenced unless it has appeared in a DIM statement. An array can appear in only one DIM statement.

```
eg.  DIM A(5,7,3),B(2),X$(3,3)
```

2.4.3 REM\BYTE

Syntax: REM\BYTE list of array names

Purpose: Specifies which arrays are byte arrays

Note 1) The arrays must be declared as numeric arrays in a previous DIM statement
2) Each byte array element occupies one byte of memory and can store a value between 0 and 255. Byte arrays have significant space and speed advantages over the default 'real' arrays.
3) The statement format was chosen so that it would be ignored by ROM PAC BASIC.

```
eg. 10 DIM A(3,3),B(7),C$(15),D(5,2)
    20 REM\BYTE A,D
    A and D arrays are specified as byte arrays.
```

2.4.4 REM\INTEGER

Purpose: Specifies which variables (scalar and array) are integer variables.

Note 1) Any arrays specified must be declared as numeric arrays in a previous DIM statement
2) Each integer scalar variable and each integer array element occupies two bytes of memory and can store a value in the range -32768 to 32767. Integer variables have significant space and speed advantages over real and string variables

```
eg. 10 DIM A(2),X(5,7,3)
    20 REM\INTEGER A,I,J,K,L2,B4
    A is specified to be an integer array and I,J,K,L2 and
    B4 are specified to be integer scalar variables.
```

2.4.5 CLEAR

Syntax: CLEAR [expression[expression]...]

Purpose: Ignored by the compiler. This statement is included to retain compatability with the ROM PAC.

2.4.6 REM

Syntax: REM [any string]

Purpose: Allows comments to be included in a program.

Note 1) A REM statement must be the last statement on a line
2) The statement is ignored by the compiler and does not generate any code.
3) REM statements may appear anywhere in a program.

2.5 ASSIGNMENT STATEMENTS

Assignment statements are used to store the value of an expression in a variable. The two assignment statements are LET and MID$

2.5.1 LET

Syntax: [LET] variable = expression

Purpose: Set a variable (scalar or array element) equal to the value of an expression.

Note 1) The keyword 'LET' is optional
2) The variable type (byte, integer, real, string) must be compatible with the expression and visa versa. If the variable is numeric, the value of the expression will be rounded to integer or converted to a real if required, before it is stored in the variable. An error occurs if the expression is outside the range of values that the variable can store. For example, an error will occur if a negative value is stored into a byte array.

2.5.2 MID$

Syntax: MID$(X$,I,J)=string expression

Purpose: Replaces part of the string variable X$ (J characters starting at the I'th position) with the value of the string expression.

Note 1) X$ is a string scalar variable or array element
2) A maximum of J characters of the string expression are stored in X$
3) An error occurs if I or J is <0 or >255
4) No characters will be stored if X$ is null or the string expression is null, or I>LEN(X$), or J=0
5) X$ is not affected by the MID$ statement

The MID$ statement has been included in the BASIC COMPILER (it is not available in the ROM PAC BASIC) because of its power and speed. MID$ is also used as a function to extract part of a string. See section 2.10.3

```
eg. 10 A$="ABCDEF"
    20 MID$(A$,3,2)="XY":REM  A$ is now "ABXYEF"
```

2.6 FLOW CONTROL STATEMENTS

2.6.1 GOTO

Syntax: GOTO line number

Purpose: To branch to a specified line and continue processing at that line.

Note If the statement at the specified line is not executable, processing continues at the next executable statement

Example: 10 GOTO 100

2.6.2 GOSUB

Syntax: GOSUB line number

Purpose: To branch to a subroutine

Note 1) The program branches to the specified line number and continues processing from there until a RETURN statement is encountered.
2) When the RETURN statement is executed the program branches back to the NEXT statement after the GOSUB.
3) When a GOSUB statement is executed the program return address is pushed onto the system stack. If more than about 10 GOSUB's are nested the Run Time Support System will return an FC ERROR.

Example: GOSUB 2010

2.6.3 RETURN

Syntax: RETURN

Purpose: The program continues execution at the statement following the GOSUB statement which called the subroutine.

2.6.4 IF - THEN - ELSE

```
Syntax:  IF expression THEN statement(s) [ELSE statement(s)]
         IF expression THEN statement(s) [ELSE line number ]
         IF expression THEN line number  [ELSE statement(s)]
         IF expression GOTO line number  [ELSE statement(s)]
```

Purpose: Allows the optional branching or execution of statements based on the value of an expression.

```
Note 1) There can be only one IF THEN statement per line.
        ie. IF A=1 THEN IF B=2 THEN C=0 is illegal.
     2) If the expression is true (ie non zero) then the THEN
        clause will be executed, or the program will branch to
        the line number in the THEN or GOTO clause.
        If the expression is false (ie. zero), the program will
        execute the ELSE clause or go to the line number
        specified in the ELSE clause. If the expression is false
        and no ELSE clause exists the program continues
        execution at the line following the IF statement.
   eg.
        10 IF A=5 THEN B=2:C=3:GOTO 50
        20 IF A<3 THEN PRINT "LESS THAN" ELSE PRINT "GREATER"
        30 IF A=4 THEN B=1:C=2 ELSE B=4:C=0
        40 IF A=1 THEN GOTO 50 ELSE 60
        50 IF A=1 THEN 70
```

2.6.5 FOR - NEXT

```
Syntax: FOR variable=exp1 TO exp2 [STEP exp3]

        [... other statements]

        NEXT [variable][,variable]...

        where variable is an integer or real scalar variable and
        exp1, exp2 and exp3 are numeric expressions.
```

Purpose: The statements between the FOR and the NEXT are executed a given number of times.

Note 1) When encountering this statement, the Compiler generates code which:

a) calculates exp1 and stores the current value of the variable.

b) calculates exp2 and exp3 (if present) and determines the number of times which the loop will be executed.

c) checks whether the number of trips through the loop is zero and by-passes the loop if so. Otherwise it will execute the loop the given number of times, regardless of whether or not the value of the variable is modified by the code within the loop. The user may, if he wishes, branch out of the loop using GOTO.

d) each time the NEXT statement is encountered, the variable will be incremented by the value of the step expression. (exp3).

2) If the step is not given , a value of 1 is assumed.

3) A significant speed advantage is gained if the variable used is an integer variable, in which case the resulting expressions will be rounded to integers. If the variable is an integer the values of the expressions must lie in the normal range for integers. ie. -32768 to 32767

4) The NEXT statement associated with a FOR statement must appear after the FOR statement in the program.

```
ie. 10 S=0:GOTO 30
    20 NEXT X:GOTO 40
    30 FOR I=1 TO 10:S=S+I:GOTO 20
```

This is not allowed by the Compiler as the NEXT precedes the FOR.

5) The maximum number of trips in a FOR-NEXT loop is restricted to 32767.

6) Only 1 NEXT may be associated with a FOR

```
ie.  10 FOR J=1 TO 10
     20 IF J<S THEN A=A+J:NEXT:GOTO 70
     30 A=A-J:NEXT
```

This is not allowed by the compiler.

7) FOR NEXT loops may be nested to a maximum depth of 10.

8) If program control jumps into the range of FOR-NEXT loop, the NEXT statement will be ignored by the Run Time Support System.

2.6.6 ON

Syntax: ON expression GOTO list of line numbers
ON expression GOSUB list of line numbers

Purpose: To allow a program to branch to one of a number of line numbers or one of a number of different subroutines. The value of the expression is rounded to an integer, n. The n'th line number is then used in the GOTO or GOSUB.

Note 1) In the case of ON - GOSUB processing will return to the statement following the ON - GOSUB statement.
2) If the expression is 0 or greater, then the number of line numbers processing continues with the statement following the ON - GOTO or ON - GOSUB statement.
3) If the expression is negative or greater than 255, an error message will be displayed.

```
Examples:  10 ON I GOTO 10,50,100
           20 ON I+A(J) GOTO 10,50,100
```

2.6.7 STOP

Syntax: STOP

Purpose: To terminate program execution. The Run Time Support System will display the available options for the user to make his choice.

Note: When execution terminates, an option menu will be displayed.

2.6.8 END

Syntax: END

Purpose: Same as STOP.

2.7 INPUT/OUTPUT STATEMENTS

Input and output statements are used to transfer data between variables and memory or variables and external devices. A number of BASIC functions also perform input/output operations (eg. POS, SPC, TAB, PEEK, INP, INKEY).

2.7.1 INPUT

Syntax:
INPUT ["prompt string";] list of variables or array elements

Purpose: To input data values from the terminal and store them in program variables.

Note 1) The ';' following the optional 'prompt string' may be replaced by a ','
2) On encountering the command, the program will display the prompt string if any, and then prompt the user for input by displaying "?"
3) If fewer values are entered than variables in the list, the program will prompt with a "?" until all values have been entered.
4) Values entered are separated by commas. Strings entered need not be enclosed by quotation marks unless they are to contain leading or trailing spaces or commas.
5) Entering carriage return only will cause the entry of a null string or a zero numeric value.
6) Entering "@" anywhere in the input line will generate a carriage return and allow the user to re-enter the line from the beginning.
7) RUBOUT may be used to delete characters
8) All other characters will be entered into the input buffer, whether ASCII or graphic, printing or non-printing except for CTRL C, which will abort the program
9) If an error is detected, the user will be prompted with the message REDO at the appropriate location.
10) The maximum line length of an input string may be 125 characters. The input buffer may be found at Hex locations F003 through F07F.

2.7.2 READ

Syntax: READ list of variables or array elements

Purpose: Obtain the next available item from the current DATA statement or, if that is exhausted, from the next DATA statement

Note Refer to the DATA statement for further details.

2.7.3 DATA

Syntax: DATA list of string and numeric constants

Purpose: To provide a list of constants to be read by a READ statement.

Note 1) READ accesses the constants in a DATA statement until the list of constants is exhausted, at which stage constants are then read from the next sequential DATA statement. The RESTORE statement may be used to revert to a previous DATA statement.

2) String constants need not be delimited by double quotes unless they contain commas or significant leading or trailing spaces.

3) An error will occur if non numeric data is READ into a numeric variable.

4) A DATA statement must be the last or only statement on a line.

5) DATA statements can appear before or after the associated READ statement.

eg.
```
100 FOR I=1 TO 5:READ X(I):NEXT I
105 DATA 20,30,40,50,60,APX,7
110 READ A$,B
```

2.7.4 RESTORE

Syntax: RESTORE [line number]

Purpose: Resets the DATA statement pointer to the first DATA statement to allow the data to be READ again. If the line number is specified, the DATA statement pointer will be set to the specified DATA statement.

eg.
```
20 RESTORE 30
30 DATA 1,2,3
```


2.7.5 PRINT and PRINT&

Syntax: PRINT [list of expressions]
PRINT& I,J [;list of expressions]

Purpose: Each expression in the list of expressions is evaluated and then displayed on the screen. In the PRINT& form, the cursor will be moved to the row and column, given by the initial two integer expressions respectively before displaying the values of the expressions.

Note 1) If PRINT is not followed by a list of expressions a CRLF will be printed.

2) A semicolon separating the items causes the expressions to appear adjacent to each other.

3) On encountering a comma in the list of expressions, the cursor will print spaces to the start of the next 14 character print zone.

4) Before printing a value, the program will check to ensure that the value will fit on the current line otherwise it will print it on the next line.

5) Numeric values printed will be followed by a space. A minus sign or a space will be printed in front of the number.

6) The second form of the PRINT statement, PRINT&, allows the user to position the cursor to the row and column determined by the two integer expressions. If I or J are real expressions, the value will be rounded to an integer. The position 1,1 (row 1, column 1) is at the top left of the screen. The second integer expression can be followed by a ',' instead of the ';'.

```
eg. 100 PRINT "SIZE=";S
    120 PRINT& 1,20;"TOP OF SCREEN"
```

2.7.6 SET

Syntax: SET I,J

Purpose: Places a 1/6 character size dot on the screen at position I,J. I is the horizontal offset from the left of the screen and must lie between 0 and 127 inclusive. J is the vertical offset down from the top of the screen and must lie between 0 and 89 inclusive. If I or J are reals, the value will be rounded to an integer.

```
eg. 100 SET 6,20
    120 SET X,Y
```

2.7.7 RESET

Syntax: RESET I, J
Purpose: Clear the dot at position I, J. This is the inverse function to SET function. See SET.

2.7.8 CLOAD*

Syntax: CLOAD* cassette unit number, numeric or string array.

Purpose: To load data from a cassette into an array.

Note 1) The cassette unit number must be a byte constant or integer scalar variable with a value of 1 or 2
2) Byte, integer, real or string arrays may be loaded.
3) The data on cassette must have been written using a CSAVE* command running under the Run Time Support System.
4) The array loaded must match in both type and number of elements with the array saved.

eg1.
```
10 DIM A(100)
50 CLOAD* 1,A
```

eg2.
```
10 DIM X$(5)
20 REM\INTEGER J
30 J=2:CLOAD* J,X$
```

2.7.9 CSAVE*

Syntax: CSAVE* cassette unit number, numeric or string array.

Purpose: To save the contents of a numeric or string array on cassette tape. Reverse operation to CLOAD*.

Note: The file is saved in the standard Exidy format, with the name of ZDATA, allowing it to be loaded with the monitor command: LO

eg.
```
10 DIM A(100),X$(5)
20 REM\INTEGER K
...
30 K=1:CSAVE* K,A
40 CSAVE 2,X$
```


2.7.10 POKE

Syntax: POKE I, J

Purpose: Stores a byte value in the given memory address.

Note 1) The low order byte of J is stored at the memory address specified by I.
2) If I or J are byte or real expressions they are converted to integer values before being used.
3) To store a byte at a location > 32767, the address should be specified as = required address -65536.
4) The PEEK function (section 2.10.4) is used to get the value of a byte in memory.

2.7.11 OUT

Syntax: OUT byte expression 1, byte expression 2

Purpose: Translates the byte given by byte expression 2 to the output port specified by byte expression 1

Note: The INP function (section 2.10) is used to get a byte from an input port.

2.7.12 WAIT

Syntax: WAIT I,J,[iexp3]

Purpose: Wait until a specified bit pattern appears at a specified input port.

Notes: The status of the port given by I is XORed with iexp3 (integer expression 3) (or zero if omitted) and then ANDed with J. The program waits until a non-zero value is returned.

eg. 100 OUT 254,11
120 WAIT 254,2,2
This will cause execution to halt until the 'RETURN' key is depressed. The OUT statement specified the section of the keyboard we wish to test. The WAIT statement waits for bit 1 become set, indicating the 'RETURN' key is being depressed.

2.8 USER DEFINED FUNCTIONS

The BASIC compiler includes a number of pre-defined BASIC functions which can be used to return a single value. In a similar way a user can define his own functions, using a single BASIC statement to define the operations of the function.

2.8.1 DEF FN

Syntax: DEF FNname(parameter)=expression
Purpose: To define a user function.

Note 1) 'name' must be a legal real variable name. 'name' prefixed by FN becomes the name of the function. 'name' cannot be used as a variable name in the program.
2) 'parameter' is a legal real variable name which is used as a dummy name in the function definition. 'parameter' can be used as a real variable elsewhere in the program and its value is not altered by the DEF FN statement or when the user function is referenced.
3) 'expression' in a numeric expression whose value (converted to a real if necessary) is returned when the user function is referenced.
4) 'expression' may not contain any variable other than 'parameter'.
5) The DEF FN statement must precede any statements which references the function which it defines.

```
eg.  10 DEF FNA(R)=3.14159265*R*R
     20 T=FNA(7)-FNA(3)
```


2.9 USER ASSEMBLY ROUTINES

The USR function allows a BASIC program to call user written assembly language routines (Routines written in Z80 machine code). Because the BASIC COMPILER is many times faster than ROM PAC BASIC, the need for assembly language routines is reduced. However, there may be situations where such routines may be required. The steps involved in using assembly language routines are:

1. Write the assembly language routine.
2. Determine a suitable position in memory to place the routine. From location 0 to FF (Hex) is not being used, unless you are running CP/M. User graphics area is programmed every time the Run Time Support System initializes and locations F003 to F07F (Hex) are also used. The user may safely choose memory above the Run Time Support System. String and temporary information area grows downward in memory from the Run TIme Support System to 100 (Hex), so this area should not be used. See Appendix C.
3. Store the routine in memory by loading it from cassette or by POKEing it into memory with program statements
4. POKE the low and high bytes of the starting address of the routine into locations 104H and 105H (260 and 261 decimal).
5. Call the routine with a USR function call with an integer argument. On entry to the routine the argument is stored in the HL register. On return from the routine the value of HL is returned as the value of the USR function.

2.9.1 USR(X)

Purpose: Calls a user assembly language routine.

Note 1) The argument X is an integer expression or a real expression that will reduce to an integer and is stored into register HL before control is passed to the users assembly language routine. The value in HL upon returning from the routine is passed as the value of the USR function. This allows the user to pass a single parameter to and from the assembly language routine. It also allows the USR function to be used in the same way as any other BASIC function such as POS or SIN.

2) Causes a call to location 103H. Location 103H contains the code for a Z80 jump. Locations 104H and 105H must contain the low and high order bytes of the address of the machine code subroutine provided by the user.

The following program POKEs a machine code program into memory and then calls it using the USR command. The example below fills the screen with the character typed on the Keyboard:

```
5 REM\INTEGER I,J,X,Y
10 DATA 125,33,128,240,17,129,240,1,0,8,119,237,176,201
15 POKE 260,0:POKE261,0
20 FOR I=0 TO 13:READ J:POKE I,J:NEXT I
30 PRINT CHR$(17):FOR I=0 TO 28:PRINT:NEXT
40 INPUT "TYPE A CHARACTER TO FILL SCREEN";A$:K=ASC(A$)
50 U=USR(K):GOTO 30
```


2.10 BASIC FUNCTIONS

The BASIC COMPILER includes a number of pre-defined BASIC functions. When a function is referenced in an expression it returns a value which is then processed like any other value in the expression.

Notation used is as follows:

I,J	any integer expression
X,Y	any real expression
X$,Y$	any string expression

Functions require one or more arguments enclosed in brackets. If an integer argument is given in place of a real argument the compiler will convert the argument to a real, and similarly where an integer is required a real number will be rounded to an integer.

2.10.1 NUMERIC FUNCTIONS

ABS(X)
Returns the absolute value of the expression, X

EXP(X)
Returns the constant 'e' raised to the power X. It is equivalent to e^X.

INT(X)
Returns the largest integer less than or equal to X. Integer division may be implemented by using a back slash instead of a forward slash. The statement: INT(R/4) is therefore equivalent to (R\4) but the latter is much faster.

LOG(X)
Returns the natural log of the expression X where X>0.

SGN(X)
Returns 1 if X>0, 0 if X=0, or -1 if X<0

SQR(X)
Returns the square root of X, where X>=0

RND(X)
Returns a random number between 0 and X. X may be negative.

2.10.2 TRIGONOMETRIC

ATN(X)
Returns the arctangent of X in radians. The result lies in the range -PI/2 to PI/2 (Where PI=3.14159)

COS(X)
Returns the cosine of the X, where X is in radians.

SIN(X)
Returns the sine of the X, where X is in radians.

TAN(X)
Returns the tangent of the X, where X is in radians.

2.10.3 STRING FUNCTIONS

String functions return integer, real or string values depending on the function.

ASC(X)
Returns the ASCII code of the first character of the string, X$. IF X$ is null, the error message 'ILLEGAL FUNCTION CALL' will result.

```
    eg. 10 PRINT ASC("B"),ASC("%")
```

CHR$(X)
Returns a one character string in which the character has the ASCII code given by the value X.

```
    eg. 10 PRINT CHR$(45),CHR$(72)
```


CVI(X$)
Functn: Converts the first 2 bytes of X$ to a 2 byte integer value. An error will occur if LEN(X$)<2. See also MKI$ for inverse operation.

```
eg. 10 PRINT CVI("AB")
```

CVS(X$)
Converts the first 4 bytes of X$ to a real number. If LEN(X$)<4 an error will occur. See also MKS$ for inverse operation.

INSTR(I,X$,Y$)
Starting at character position I in X$, INSTR searches for the first occurrence of Y$ in X$. INSTR returns the values 0 if X$ is null or if no match is found or if I>LEN(X$). INSTR will return I if Y$ is null. Otherwise n is returned where n is the position of the first character of X$ where a match was found. The first character position of X$ is 1. An error occurs if I<0 or I>255.

```
eg 10 A$="ABCDEF":B$="CD":C$="CE"
   20 PRINT INSTR(1,A$,B$),INSTR(2,A$,B$),INSTR(1,A$,C$)
   This will print:  3  3  0
```

LEFT$(X$,I)
Returns the I characters to the left of X$

LEN(X$)
Returns the length of the string expression, X$

MID$(X$,I,J)
Return a substring of X$, of length J characters, starting at the I'th character of X$. An error occurs if I or J are <0 or >255. A null string is returned if J=0 or if I>LEN(X$) or if X$ is null. If J is greater than the number of characters available starting at the I'th character then only the available characters will be returned.

```
eg. 10 PRINT MID$("ABCDE",2,3)
    BCD
```


MKI$(I)
Returns the 2 byte string representation of the two byte iexp1. See also CVI.

MKS$(X)
Returns the 4 byte string representation of the real expression, X. See also CVS.

RIGHT$(X$,I)
Returns a string comprising the rightmost I characters of X$. If I)=LEN(X$) then all of X$ is returned. A null string is returned if X$ is null or I=0. An error occurs if I<0 or I>255.

```
eg. 10 PRINT RIGHT$("ABCDE",3)
    CDE
```

SPC(I)
Returns a string of spaces of length I. See also section 2.10.4

STR$(X)
Returns the string representing the value of X. This consists of the ASCII characters which would be printed if X were to be printed. The first character will be either a space or minus sign. See VAL for inverse operation.

VAL(X$)
Returns the value which is the numeric equivalent of the string, X$. See also STR$ for inverse operation. The number is terminated by the first non-numeric character of the string.if the first non blank character of X$ is not +,- or a digit the return value will be 0.

```
eg. 10 PRINT VAL("0,53ABC"),VAL("ABC")
    0.53  0
```


2.10.4 INPUT/OUPUT FUNCTIONS

INKEY
Returns the integer equivalent of the ASCII code of the key depressed on the keyboard. If no key has been depressed a zero is returned. If a key is depressed, processing is halted until the key is released. INKEY is not available in ROM PAC BASIC.
eg. J=INKEY

INP(I)
Returns the integer value of the byte from port I, where I is in the range 0-255

PEEK(I)
Returns the value of the byte at memory address I. To PEEK at an address higher then 32767, the value 65536 must be subtracted from the address. For example, to PEEK at address 65535, PEEK(-1) would be used.

POS(I)
Returns the integer value of the current print position. The first character on a line is at position 1. I is a dummy argument.

SPC(I)
Returns a string of length I containing all spaces.

```
eg. 100 PRINT SPC(10);
    110 A$=SPC(20)
```

Unlike all other functions that return a string, SPC(I) does not have a '$' before the first bracket character.

TAB(I)
Prints spaces to the I'th print position on the line. Ignored if the print position is greater than I.

2.10.5 GENERAL

FRE(I)
Returns the number of bytes of memory that are not being used by the BASIC ROM PAC or the user program. The value represents the number of free bytes in string space before compaction. I is a dummy argument.

CHAPTER 3. PROGRAM DEVELOPMENT

3.1 DIFFERENCES BETWEEN ROM PAC BASIC AND BASIC COMPILER

Every effort has been made to allow any ROM PAC BASIC program to run with the BASIC COMPILER. However, because a compiler operates differently from an interpreter, some restrictions had to be placed on the ROM PAC BASIC language. It is unlikely that these restrictions will seriously inconvenience any user.

The BASIC COMPILER includes additional powerful features which save space, increase speed and provide additional programming capability. Suggestions in how to make the best use of the additional features are contained in section 3.2.

3.1.1 RESTRICTIONS ON THE USE OF THE ROM PAC BASIC LANGUAGE.

a) All arrays must be dimensioned explicitly and may have a maximum of 3 dimensions.
b) All dimension statements must appear at the beginning of the program, hence variables may not be used to assign the dimensions.
c) No direct commands are available when the compiler is running, unless the user returns to BASIC.
d) CLOAD* and CSAVE* require a ',' between the unit number and the array name. The unit number must be specified either as a constant or an integer variable.
e) Only one NEXT statement is allowed for each FOR statement and the NEXT statement must appear in the program after the FOR statement.
f) Only one IF THEN statement may appear on a program line.

3.1.2 ADDITIONAL FEATURES AVAILABLE IN THE BASIC COMPILER

a) Byte and integer constants to save space and increase speed
b) Integer variables and byte and integer arrays to save space and increase speed.
c) An integer divide operator "\" to increase speed and XOR operator to provide extra programming power.
d) Enhanced string handling methods to increase speed and reduce the occurrence of compaction of the string space.
e) The MID$ assignment statement to increase speed and avoid string space compaction.
f) The powerful IF - THEN - ELSE construction to simplify programming
g) The PRINT& statement to position the curser anywhere on the screen before printing
h) Graphics statements (SET and RESET) to turn on or turn off a 1/6th of a character dot on the screen.
i) The CLOAD* and CSAVE* statements can utilize any type of array, including string arrays. The array is saved in the standard monitor format, allowing the user to load with either CLOAD* or the monitor command LO.
j) RND(X) returns a real number between 0 and X. ROM PAC BASIC returns a real between 0 and 1.
k) Additional BASIC string functions, CVI, CVS, MKI$, MKS$ and INSTR are provided.
l) The INKEY function allows the user to check if a key is being depressed without having to write and call a machine code routine.
m) The USR function allows the passing of a parameter to and from the machine code routine. It also is implemented like other functions and may be used in statements such as: PRINT USR(4)
n) SPC returns a string. Thus PRINT SPC(I) works in the same way as in ROM PAC BASIC, but in the same way as all other functions. It may be used in a general way: eg. A$=SPC(5).

3.2 OPTIMIZING PROGRAM PERFORMANCE

The BASIC COMPILER has been designed to generate very compact code which executes as fast as possible. This section describes programming techniques which can be used to take full advantage of the optimizations done by the compiler.

3.2.1 USE OF BYTE AND INTEGER CONSTANTS AND VARIABLES

Because they require much less space and provide for much faster program execution, bytes and integers (constants and variables) should be used whereever possible, in preference to reals and strings.

Even greater savings in space and time will be achieved if the user is careful to structure his statements in the formats explained below.

In the following description, the variables v, v1, and v2 will be used to represent any or all of the following:

1) A byte constant eg. 0,1,2,67,255
2) A positive integer constant eg. 0,256,500,32767
3) A integer scalar variable eg. I, A3 where I, A3are integers.
4) A special form of an array. The array must be a byte or integer array with one dimension and the subscript must be a single integer scalar variable eg. A(J), K4(L1) where A and K4 are byte or integer arrays and J and L1 are integer scalar variables.

OPTIMIZED (FASTEST AND SHORTEST) FORMS OF THE LET STATEMENT

FORM	EXAMPLES
v=0	10 I=0:A(I)=0
v=1	10 I=1:A(I)=1
v1=v2	10 I=32000:A(I)=J
v1=v1+1	10 I-I+1:A(I)=A(I)+1
v1=v1-1	10 I=I-1:A(I)=A(I)-1
v1=v2 opr v3	10 I=J+425:A(I)=B(K)*J

where opr is one of: +, -, *, \, /, =, <>, <, >, <=, >=

OPTIMIZED IF - THEN STATEMENT

The forms of the optimized IF statement are:

```
IF v1 relopr v2 THEN
IF v1 relopr v2 GOTO
```

where relopr is one of: =, <>, <, >, <=, >=

Examples:

```
10 IF I=32 GOTO 110
20 IF A(I)<1600 THEN PRINT I
30 IF A(I)<=B(J) THEN 1000
```

OPTIMIZED FOR - NEXT STATEMENT

The form of the FOR - NEXT statement which executes the fastest is:

```
FOR integer scalar variable = expression1 TO expression2
```

Note that the index is an integer scalar variable and that STEP is not included. The expressions will be rounded to integers.

Examples:

```
10 REM\INTEGER I,J
20 FOR I=1 TO 1000:J=I:NEXT
30 FOR I=J TO A(I)+4:J=A(I):NEXT J
```

OPTIMIZED STRING OPERATIONS

The BASIC COMPILER provides two methods for speeding up string operations. The first method is used to avoid string compaction and the second method provides for very fast replacement of parts of a string.

AVOIDING STRING COMPACTION

The BASIC COMPILER allocates string space dynamically. Thus space is allocated to string variables only when required. However, if a string is reduced in length, a block of unused space is created. This block of space must be recovered by a process called compaction. Similarly if a string is increased in length the string will no longer fit into its existing space and either all strings below it must be moved to make space or else new string space is allocated and a block of unused space is created. This space will need to be compacted at some stage. Because compaction can take a long time, it is best to avoid it if possible.

The BASIC COMPILER stores strings in such a way that compaction can be reduced or even avoided completely. In addition to storing the length of each string (eg. LEN(X$)) the compiler also stores the actual space allocated by the string. Thus, if each string is initially set to the largest length it will occupy during execution, then compaction will never be required, even if the length of the string varies up and down. If it is not possible to set each string to its maximum execution length then compaction may eventually occurr, as strings are increased and reduced in length and blocks of unused space are created. However, the number of times compaction occurs should be reduced.

To set a string to its maximum execution length it is not sufficient to set the string to a string constant or to a variable which has been set to a string constant. The reason is that the string constants are embedded in the compiled code and if a variable is set equal to a string constant, the variable does not occupy any space, it simply points to the string constant. However, if two string constants are concatenated or if a string function is called, a new string is created and a string variable can be set equal to this new string. The simplest way of setting a string to its maximum execution length is to use the SPC function.

A string can be set to its maximum length in the following ways:

```
10 A$=SPC(20)
20 A$="                    "+"                    "
```

The first statement using the SPC function is recommended because it takes much less space in the compiled code.

In the following example, compaction will never occur.

```
10 REM\INTEGER I
20 A$=SPC(200):REM SET MAX LENGTH TO 200
30 A$="":REM SET LEN(A$)=0
40 REM NOW INCREASE LEN(A$) IN STEPS UP TO 200
50 FOR I=1 TO 200:A$=A$+"A":NEXT I
60 GOTO 30:REM REPEAT THE PROCESS
```

SUBSTRING REPLACEMENT

Suppose string A$ is 8 characters long and it is required to replace characters 4 to 6 with a new string X$ of length 3. The normal way to do this is as follows:

```
10 A$=LEFT$(A$,3)+X$+RIGHT$(A$,2)
```

A neater and much faster alternative is using the left hand MID$ statement:

```
10 MID$(A$,4,2)=X$
```

The MID$ statement can also be used to replace the first or last groups of characters of a string by setting appropriate values for the MID$ arguments.

GENERAL HINTS TO OPTIMIZE PROGRAM PERFORMANCE

The following programming techniques may be useful in reducing program size, increasing execution speed or to simplify program complexity.

1) Not outputing line numbers in the compiled code. See OPTION.
2) Using the IF - THEN - ELSE form if the IF statement
3) Using PRINT& to position the cursor.

3.3 APPLICATION EXAMPLES

3.3.1 PRINTER INTERFACE

If a printer is connected to the Sorcerer, the user may print the characters appearing on the screen by:

a) using the monitor SE O=L command if the printer is Centronics or
b) setting the output to an assembly language routine with the SE O=XXXX command.

The same result may be achieved with the use of the following BASIC routine:

```
100 REM  PRINTER ON:  H=233, L=147
110 REM  PRINTER OFF: H=224, L=27
120 T=PEEK(-4096)+256*PEEK(-4095)-46
130 IF T>32721 THEN T=T-65536
140 POKE T,H:POKE T-1,L
```

The above program finds the top of RAM, calculates the address of the output vector in the MWA and POKEs in H and L. H and L must be set to 233 and 147 respectively, if it is to substitute for the monitor SE O=L command. If a special driver is being used, different values for H and L must be employed. H is the high order byte of the address of the assembly language routine while L is the lower. This routine may be called as a subroutine from within a BASIC compiled program.

3.3.2 USING THE COMPILER WITH DISKS

A) USING DISKS WITH THE COMPILER

The Compiler and compiled code may be used together with disks in a limited way assuming the user has:

1) A floppy disk drive system running CP/M
2) More than 32K RAM

It is an advantage to have the disk controller set up so that the user may boot the disk drive with the BASIC ROM PAC inserted. Often the disk drive uses the same memory as the ROM PAC as DMA area. MICROPOLIS, VISTA and FDS drives may be modified to use another area.

If the disks and ROM PAC can be used at the same time, the user may wish to save the compiler onto disk and eliminate the need to load it from cassette. This may be accomplished by:

1) Insert the BASIC ROM PAC and turn on the Sorcerer
2) Save the BWA by typing:
```
BYE
MO 100 200 FE00
```
3) Boot the CP/M disk. The CP/M should sit as high in memory as possible and should be well above 32K so that it does not use memory below 8000 (Hex).
4) Enter the monitor by typing:
```
DDT
GE003
```
5) Move back the BWA by typing: `MO FE00 FF00 100`
6) Now LOG the 32K version of the Compiler from cassette. When loaded the compiler will give a CN ERROR because no BASIC source is available to compile.
7) Type BYE to re-enter the monitor and then give the monitor command GO 0 to Warm Start CP/M.
8) Save the compiler on disk with the CP/M command:
`SAVE 127 COMP.COM`

To load the compiler from disk, simply boot the disks with the BASIC ROM PAC inserted and type: COMP. The compiler will load, give a CN ERROR and return you to BASIC, ready for compilation. Your BASIC program may either be typed in or CLOADed at this stage.

It is interesting to note that if a BASIC source program was in memory at step 7 above, it would have been saved along with the compiler and would be loaded back into memory at this stage, ready for compiling or continued de-bugging.

B) USING DISKS WITH THE COMPILED MODULE

It is not necessary to have the ROM PAC inserted in order to use the LGO module. If the LGO module has been developed with CP/M as explained above, it may be saved on disk by following the above instructions from point 7. The only difference being 1) Enter the monitor with Option 2 (Displayed by the Run Time Support System) after creating the LGO module and 2) choosing a different name. The extension should still be .COM.

The module can then be loaded and executed at any time, with or without the ROM PAC being insterted.

C) PASSING DATA TO AND FROM DISK

Data may be saved and retrieved from disk in a limited way from within a running LGO module. Simply have the LGO module POKE data into memory between 32K and the bottom of CP/M before ENDing and then return to CP/M by typing GO 0. The data and LGO module may be saved on disk with the same CP/M command used to save the LGO module originally. The data may then be PEEKed out of memory on the next run. Note that HEX location 1B7 and 1B8 contain the pointer to the lowest available memory which the compiler will use. This is normally set to 1D5, but may be changed to point to higher memory in order to reserve some memory for the user. The user should not corrupt memory from 0000 to 01D5 HEX.

3.3.3 FULL EXAMPLE IN USING THE COMPILER

The following is an example in using the compiler with graphics. Firstly turn on the Sorcerer with the BASIC ROM PAC inserted then type in the following program:

```
5 DIM S(20),T(20)
8 REM\BYTE S,T
10 REM\INTEGER X,Y,A,B,C,D
13 D=10
15 PRINT CHR$(12);CHR$(1);
20 X=RND(30)+3:Y=RND(30)+3:A=1:B=1
30 C=C+1:IF C=D THEN C=0
35 RESET S(C),T(C)
40 SET X,Y:S(C)=X:T(C)=Y
50 IF X=127 THEN A=-1
55 IF X=1 THEN A=1
60 IF Y=89 THEN B=-1
65 IF Y=1 THEN B=1
70 X=X+A:Y=Y+B
80 IF INKEY=0 THEN 30
```

Now type BYE and LOG the compiler. After loading, the compiler will execute and compile the above BASIC program. The compiling process will take 8 seconds. When compiling is complete, a menu will be displayed. If an error was displayed, choose option 3 to return to BASIC, fix the error, type BYE and then type GO 100 to re- compile.

If no error was found, you may now choose option 1, to execute the compiled code. The screen should clear and a worm move around the screen. To abort execution, hit any key, whereby the menu will again be displayed. Choose option 3 to return to BASIC and change line 10 to D=20 and then type BYE followed by GO 100 to re-compile. Execute the compiled code again and notice the difference to the length of the worm.

You may now choose to create a LGO stand-alone module. This procedure creates a machine code program that can execute without the compiler or BASIC ROM PAC in memory. Choose option 4. A new menu will now be displayed. Choose to return to the Exidy monitor and note that the start and end locations of the stand-alone module are printed. You may execute the program by typing GO 100 and/or save the program again on cassette or disk.

CHAPTER 4. COMPILER OPERATING INSTRUCTIONS

The following steps are followed in order to compile a program.

1) Load the BASIC ROM PAC and turn on the power.
2) Load the source program from the cassette if it is already saved, else enter it from the keyboard. Debug the program in the normal way by using the ROM PAC. The user is advised to save the BASIC source on cassette before compiling.
3) Return to the monitor with the BASIC command: BYE
4) Load and execute the compiler with the monitor command: LOG. The source program will be compiled and the resulting object code will be stored in memory above the BASIC source code. If errors are encountered, error messages will be displayed on the screen. The errors must be corrected before running the compiled code. This is explained below.

When the compilation is complete, the compiler presents the user with a menu and the choice of returning to the monitor, returning to ROM PAC BASIC to modify or correct the source program, running the program which has just been compiled or creating a 'Load and Go' (LGO) module. This module consists of the object code produced from the compilation and the run time support system. It may be saved on cassette for subsequent execution.

If errors are detected, the user may choose the option to return to BASIC, in which case the compiler will execute a Warm Start to BASIC. The source program is still in memory and available for the user to modify. The monitor stack and BASIC string stack have been moved below the compiler and Run Time Support System to ensure that the user may edit or run the program within the ROM PAC BASIC system without affecting the area of memory occupied by the compiler. When the program is again ready to compile, the user returns to the Exidy monitor by typing BYE then enters the compiler by typing GO 100. Again, the user is advised to save the BASIC source.

If compilation was successful, the user may wish to choose the option of running the compiled code. It is possible to interRupt the program while it is executing by entering CTRL C or RUNSTOP. The former aborts the program and again presents the menu while the latter temporarily halts execution until any other key is hit. At the conclusion of the run, the menu will be presented.

While it is possible to return to the Exidy monitor (with a warm start) directly, the user is cautioned that before saving the compiled program on cassette, it is necessary to create a 'Load and Go' module. When the user chooses to create such a module, the compiled object program is moved below the Runtime Support System to overlay and replace the compiler. Once this is done, the options available to the user are running the 'LGO' Module, returning to the Exidy Monitor, or saving the module on cassette.

If the user decides to run the 'LGO' module, the Run Time Support System assumes that all memory before it is available for data. The original Basic source program may well be overwritten at this stage, and the user is advised to store the source code on cassette prior to compilation. Again, the user may abort the run or pause by typing CTRL C or RUNSTOP respectively. At the end of execution, the Run Time Support System will again display the three options.

When the user is satisfied with the compiled program he may save the 'LGO' module on cassette by choosing the appropriate option. After saving the program, the Run Time Support System will again display 3 options. The user may choose to return to the Exidy Monitor, whereby the starting and ending addresses of the memory occupied by the module and the GO address are displayed. The GO address for LOG execution is automatically entered into the Monitor Cassette header block. The user will then save the module using the monitor command SA.

APPENDIX A - ERROR MESSAGES

Error messages at compilation

CE	Expression too complex
DD	Redefined array
DL	Duplicate line number. See note below
IC	Illegal character in BASIC program
IF	Illegal function name or error in function definition
IV	Illegal Variable
LO	Line too complex. Divide into two separate lines
MI	More than one IF statement on a line
NF	Unmatched FOR or NEXT
NL	Line number missing or error
RC	Error in real constant definition
SN	Syntax error
SO	Statement out of order
ST	String formula too complex
TM	Type mismatch
TO	User symbol table overflow
UF	Undefined user function
UL	Undefined line number referenced

Error messages at execution

BS	Bad subscript
CN	Can't continue. No program or continue after error
FC	Illegal function call. Possibly caused by: a) LOG with argument <=0, b) SQR with argument <0, c) A to the power of B with A<0, d) call to USR with address of assembly language, e) routine not set, f) parameters outside allowed range, g) RET without a GOSUB
LS	String too long
OD	Out of data
OM	Out of memory- program and data too large. Try creating an LGO module
OS	Out of string space
OV	Overflow
/0	Divide by zero occurred

Note: If an UL error is found during compilation, several internal counters are corrupted and DL errors will be printed. Simply correct the UL error and re-compile.

APPENDIX B - RESERVED WORDS

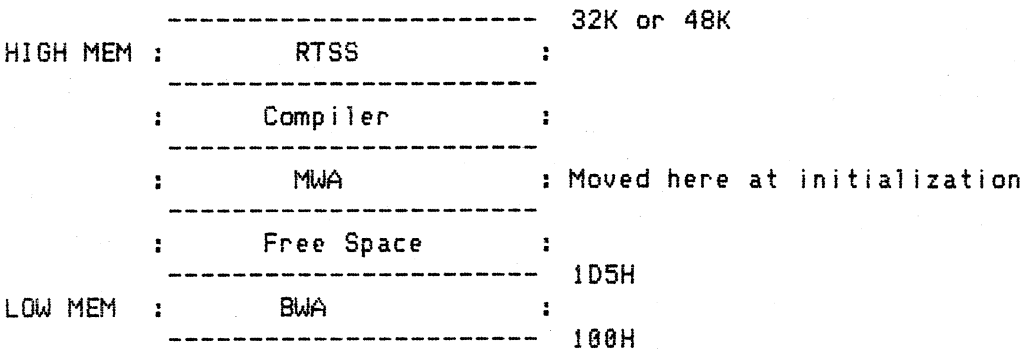
ABS	AND	ASC	ATN	BYTE	CHR$	CLEAR
CLOAD*	COS	CSAVE*	CVI	CVS	DATA	DEF
DIM	ELSE	END	EXP	FN	FOR	FRE
GOSUB	GOTO	IF	INKEY	INP	INPUT	INSTR
INT	INTEGER	LEFT$	LEN	LET	LOG	MID$
MKI$	MKS$	NEXT	NOT	ON	OPTION	OR
OUT	PEEK	POKE	POS	PRINT	READ	REM
RESET	RESTORE	RETURN	RIGHT$	RND	SET	SGN
SIN	SPC	SQR	STEP	STOP	STR$	TAB
TAN	THEN	TO	USR	VAL	WAIT	XOR

APPENDIX C - COMPILER MEMORY MAPS

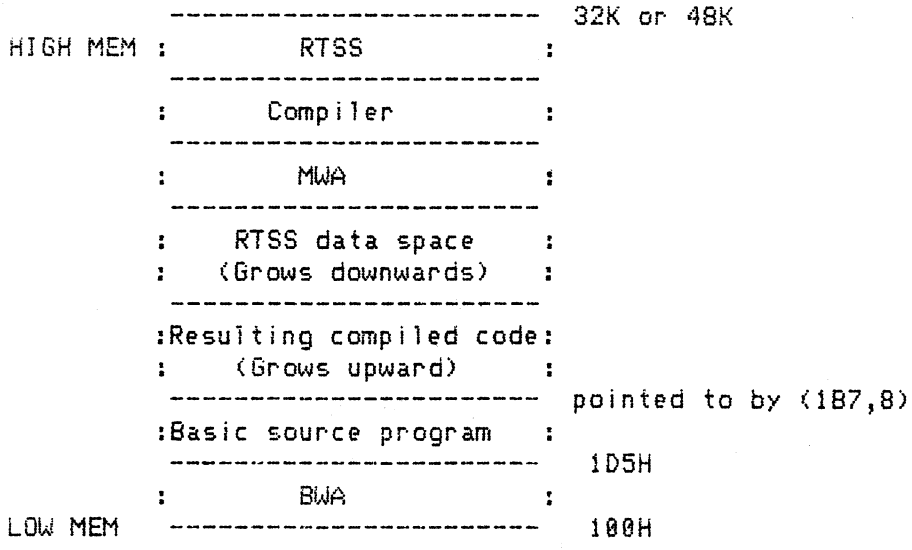
1. Notation: RTSS - Run Time Support System
LGO - Load and Go module
MWA - Exidy monitor work area
BWA - BASIC ROM PAC work area

Note that the Compiler is a compiled BASIC program running under the RTSS.

2. Memory map immediately after compiler is loaded and initialized and where no BASIC source is present.

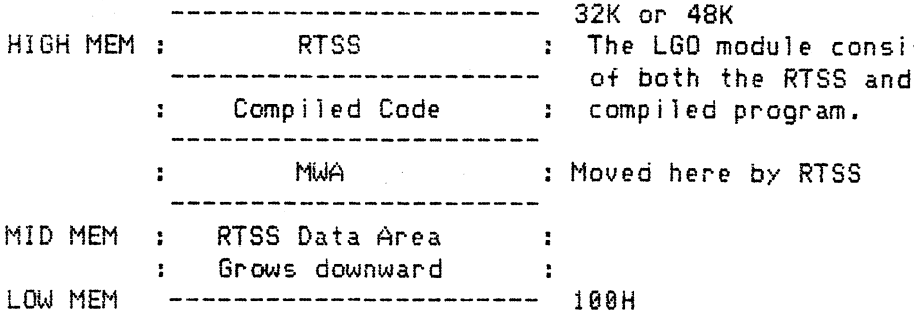


1. Memory map while compiling, or while a compiled program is running (before an LGO module has been created).



NB. Diagrams not to scale.

3. Memory map after LGO module has been created or is executing.



4. On return to Exidy monitor after creating a 'Load and Go' module.