M6809BASICM(D2)

JUNE 1982

BASIC-M

INTERPRETER / COMPILER

User's Guide

Second Edition

1. INTRODUCTION 01-01

1.1 BASIC-M - Introduction 01-01
1.2 BASIC-M unique features 01-01
1.2.1 Implementation 01-01
1.2.1.1 Compiler / Interpreter 01-01
1.2.1.2 Object code and runtime characteristics . . . 01-01
1.2.1.3 Edit capabilities and error detection 01-01
1.2.2 Data types and address assignment control. 01-02
1.2.2.1 Variable names 01-02
1.2.2.2 Data types 01-02
1.2.2.3 Data structures 01-02
1.2.2.4 Address assignment 01-03
1.2.3 Language extensions 01-03
1.2.3.1 Interrupt and condition monitoring 01-03
1.2.3.1.1 Interrupt monitoring 01-03
1.2.3.1.2 Condition monitoring 01-04
1.2.3.2 Data formatting 01-04
1.2.3.2.1 Formatting output data 01-04
1.2.3.2.2 Editing memory-resident data 01-06
1.2.4 Matrix operations 01-06
1.2.5 Disk I/O 01-07
1.2.6 Built-in functions 01-07
1.2.7 User-defined functions / procedures 01-08
1.2.8 Assembly-language interface 01-08

2. PROGRAM ORGANIZATION AND ELEMENTS 02-01

2.1 Statement lines 02-01
2.2 Statement types 02-02
2.2.1 Input/output statements 02-02
2.2.2 Arithmetic and string processing statements 02-03
2.2.3 Control statements 02-03
2.2.4 Nonexecutable statements 02-03
2.3 Statement composition 02-04
2.3.1 Character set 02-04
2.3.2 Keywords 02-06
2.3.3 Variable / procedure names 02-06
2.3.4 Constants 02-07
2.3.4.1 Literal constants 02-07
2.3.4.2 Numeric constants 02-08
2.3.4.2.1 Decimal constants 02-08
2.3.4.2.2 Hexadecimal constants 02-08
2.3.5 Expressions 02-09
2.3.5.1 Arithmetic expressions 02-09
2.3.5.2 Literal expressions 02-10
2.3.5.3 Relational expressions 02-10
2.3.5.4 Logical expressions 02-11
2.3.6 Codes 02-12

3. DATA TYPES AND STRUCTURES 03-01

3.1 Standard BASIC data types 03-01
3.1.1 Character data 03-01
3.1.2 Real data 03-02

3.2 Non-standard BASIC data types 03-03
3.2.1 Byte data 03-03
3.2.2 Integer data 03-04
3.3 Access to individual bits 03-04
3.4 Mixing data types in expressions 03-05
3.5 Data structure : the array 03-06
3.5.1 General description 03-06
3.5.2 Array declaration 03-07
3.5.2.1 Declaring byte and integer arrays 03-07
3.5.2.2 Declaring real and character arrays 03-08
3.5.3 Arrangement of arrays in storage 03-09
3.5.4 Matrix-oriented statements 03-09
3.5.5 Matrix re-sizing 03-09
3.6 Variable and array address control. Equivalence 03-10

4. EXPRESSIONS 04-01

4.1 Rules for writing arithmetic expressions 04-01
4.1.1 Separation of constants and variables 04-01
4.1.2 Separation of arithmetic operators 04-01
4.2 Order of evaluation of arithmetic expressions 04-02
4.3 Mixed-mode arithmetic expressions. Impact on size and speed. 04-03
4.4 Data types produced by arithmetic expressions 04-04
4.5 Literal expressions 04-05
4.6 Evaluation of logical expressions 04-06

5. BASIC-M SIMPLE STATEMENTS 05-01

5.1 The LET statement 05-01
5.2 The REM statement 05-01
5.3 The READ , DATA , and RESTORE statements 05-02
5.4 The console INPUT statement 05-03
5.5 The PRINT statement - simple form 05-06
5.6 The DIGITS and LINE statements 05-08
5.7 The GOTO statement 05-09
5.8 The conditional GOTO statement 05-10
5.9 The GOSUB and RETURN statements. 05-11
5.10 The conditional GOSUB statement 05-11
5.11 The IF statement 05-11
5.12 The FOR and NEXT statements 05-12
5.13 The STOP , PAUSE , and END statements 05-14
5.14 Illustrative examples 05-15

6. PRINT USING 06-01

6.1 General description 06-01
6.2 Format descriptors 06-03
6.2.1 The integer descriptor 06-03
6.2.2 The string descriptor 06-03
6.2.3 The hexadecimal descriptor 06-04
6.2.4 The horizontal spacing descriptor 06-05
6.2.5 The vertical spacing descriptor 06-05
6.2.6 The fixed-point descriptor 06-05
6.2.7 The exponential descriptor 06-06
6.2.8 The commercial descriptor 06-07

7. DECLARATION STATEMENTS 07-01

7.1 Declaring byte variables 07-01
7.2 Declaring integer variables 07-02

7.3 The DIM statement 07-02
7.4 Declaring external subroutines 07-03
7.5 Runtime initialization 07-04

8. REAL-TIME MONITORING 08-01

8.1 The ON interrupt THEN statements 08-01
8.2 The ON KEY statement 08-03
8.3 The WHEN ... THEN statement 08-04
8.4 The ON ERROR statement 08-09
8.5 The NEVER statements 08-10
8.6 More about the RETURN statement 08-11

9. MATRIX OPERATIONS 09-01

9.1 The classical approach 09-01
9.2 The MAT READ statement 09-02
9.3 The console MAT INPUT statement 09-02
9.4 The MAT PRINT statement 09-04
9.5 Copying a matrix 09-05
9.6 Matrix addition and subtraction 09-07
9.7 Matrix multiplication 09-08
9.8 Scalar operations 09-09
9.9 Identity matrix 09-10
9.10 The MAT SET statement 09-11
9.11 The MAT ZER and MAT CON statements 09-11
9.12 Matrix transposition 09-12
9.13 Matrix inversion 09-12

10. FUNCTIONS AND PROCEDURES 10-01

10.1 User-defined functions 10-02
10.2 Procedures 10-05
10.3 Assembly-language interface 10-08

11. BUILT-IN FUNCTIONS 11-01

11.1 Trigonometric functions 11-01
11.2 Other mathematical functions 11-01
11.3 Logical functions 11-02
11.4 String functions 11-02
11.5 Miscellaneous functions 11-03
11.6 Default type of the argument 11-03
11.7 Illustrative examples 11-05

12. DISK FILE INPUT / OUTPUT 12-01

12.1 General description 12-01
12.1.1 File types 12-01
12.2 The OPEN statement 12-05
12.3 The file INPUT statement 12-06
12.4 The end of file test 12-09
12.5 Output transfer to file via the PRINT statement 12-11
12.6 The REWIND statement 12-12
12.7 The CLOSE statement 12-12
12.8 Alphanumeric access key 12-13
12.9 Array input/output with disk files 12-14
12.9.1 Input of an array from a disk file 12-14
12.9.2 Output of an array to a disk file 12-15

13. SYSTEM COMMANDS 13-01

13.1 Operating Modes 13-01
13.2 Invoking BASIC-M 13-01
13.3 Interpreter Mode 13-04
13.3.1 Creating the source program 13-04
13.3.2 Auto line-numbering 13-04
13.3.3 RESEQuence 13-05
13.3.4 LIST and LIST Erroneous statement lines commands . 13-05
13.3.5 FLAGON and FLAGOFF commands 13-06
13.3.6 The DELete Command 13-06
13.3.7 The RENAME command 13-07
13.3.8 Returning to the disk-operating system 13-07
13.3.9 The RUN command 13-08
13.3.10 The TRON and TROFF commands 13-08
13.3.11 The PATCH command 13-09
13.3.12 The NEW command 13-09
13.3.13 The COMPILE command 13-09
13.4 Compiler Mode. 13-11
13.5 The BLOAD Utility. 13-13
14. PERFORMANCE CHARACTERISTICS 14-01

14.1 Requirements 14-01
14.2 Space estimates 14-01
14.3 Speed estimates 14-02

APPENDICES

A. ASCII Character Set A-01

B. Syntax Error Codes / Messages B-01

C. Compilation Error Codes / Messages C-01

D. Runtime Error Codes / Messages D-01

E. Summary of BASIC-M Statements and Functions E-01
F. Chaining the Execution of Disk Resident Overlays F-01
G. Partitioning a BASIC-M Source Program G-01
H. Supplementary Manual: Kernel Requirements H-01
I. Minimum Kernel for MM19: Program Listing I-01
J. BASIC-M Example Program for MM19 . . . . J-01

CHAPTER 1

1.1 INTRODUCTION

This chapter is an introduction to BASIC-M. It gives an overview of the language and its implementation characteristics. The reader already acquainted with BASIC is especially invited to read through this section.

1.2 BASIC-M unique features

1.2.1 Implementation

1.2.1.1 Compiler / Interpreter

In interpreter mode, the operator interacts with his source program which is held in the system RAM. The "RUN" command causes the source to be compiled into an object code which is immediately executed (high-speed compilation : approximately 50 lines/second) under control of the Runtime Package.

In compiler mode, the object data can overwrite the compiler and possibly the BASIC source program; thus no further interaction is possible, but more memory space is available at execution time.

In either mode, an option exists to force the compiler to produce a compact code (five bytes less per statement line); this option however, precludes further tracing or monitoring (see "WHEN ...THEN" and "ON ERROR THEN" statements).

1.2.1.2 Object code and Runtime Package

Both the Runtime Package and the code produced by the compiler are position-independent, a powerful feature derived from the MC6809 processor. The Runtime installation address in the end system can be specified at compilation time.

The scratchpad RAM allocated to the BASIC variables and stacks can also be easily controlled, either by type declaration statements, or when invoking the compiler.

1.2.1.3 Edit capabilities and error detection

Several system commands exist for automatic line numbering, resequencing and for renaming variables or user-defined functions or procedures.

Errors are detected at three levels :

- when entering the source (syntax errors)
- when compiling (compile-time errors)
- when executing the compiled code (runtime errors ; these may be optionally processed by the user program).

1.2.2 Data types and address assignement control

1.2.2.1 Variable names

Unlike standard BASIC, BASIC-M accepts multicharacter variable and user-defined function or procedure names. This allows better readability and program maintenance. The "RENAME" system command provides a means to upgrade standard BASIC variable names. Thus, for instance, M$ may be easily changed to Month$, T(2) to Time_of_the_day(2), ... and so on.

1.2.2.2 Data types

The following four types are supported :

- Real : 5-byte data in a format allowing a dynamic range of E+38 to E-38, with an accuracy of over 9 digits.
- String : 31-character variables.
- Byte : unsigned 8-bit data.
- Integer : signed 16-bit data.

A variable is assigned one of the above types either implicitly (real and string variables conform to BASIC conventions), or explicitly (type declaration via the BYTE or INTEGER statements).

In addition, a single bit can be easily accessed within a byte or integer, just by specifying its position within the variable. Thus the testing or setting of one bit in high-level language becomes a simple matter, as illustrated below.

Examp le 1.0

```
10 BYTE Pia ADDRESS $8008  \  8008 is an hex constant
20 REM reset bit #0 if bit #7 is set
30 IF Pia[7]=1 THEN Pia[0]=0
```

1.2.2.3 Data structures

BASIC-M supports one and two-dimensional arrays that may contain elements of either of the four data types mentioned before.

1.2.2.4 Address assignment

Unless otherwise notified, the compiler takes care of allocating storage to the program variables. However, the user may force the assignment of absolute base addresses to some of his program variables or procedures (external assembly language subroutines). This is achieved via the ADDRESS keyword. For instance, the following statements will define a two-dimensional byte matrix based at the hexadecimal location $C000, and a subroutine starting at hex address $F024.

Example 1.1

```
12  BYTE Alpha_memory(22,80) ADDRESS $C000
15  EXTERNAL Pdata ADDRESS $F024
18  INTEGER Graphic(22,40) ADDRESS Alpha_memory
```

The possibility of achieving the effect of the FORTRAN "EQUIVALENCE" statement is also illustrated in the previous example : the arrays labelled Alpha_memory and Graphic respectively, occupy the same memory space, although they are not of the same type.

1.2.3 Language extensions

1.2.3.1 Interrupt and condition monitoring

BASIC-M permits the user to work at a "low level", i.e close to the machine environment.
The data types and addressing, as described in the previous paragraph, are a first step to meet this requirement. The statements that are briefly discussed next take it a step further. They all allow for an easy monitoring of external events or conditions.

1.2.3.1.1 Interrupt monitoring

Eight statements are provided for processing interrupt requests :

```
ON NMI THEN ...
ON IRQ THEN ...
ON FIRQ THEN ...
ON KEY key list THEN ...
```

and their counterparts :

```
NEVER NMI
```

```
NEVER IRQ
NEVER FIRQ
NEVER KEY key list
```

The first three statements of each series refer to the MC6809 interrupt sources, while "ON KEY" and "NEVER KEY" refer to the management of function keys. Below are examples to be described in more detail further on in this manual.

Example 1.2

```
100  ON NMI THEN Update_time    \ Real-time clock
320  ON FIRQ THEN GOSUB 480
480  NEVER KEY 3,5,12
```

1.2.3.1.2 Condition monitoring

One of the nicest statements in BASIC-M is the:

"WHEN condition THEN action" statement.

It differs in many respects from the standard "IF ... THEN" statement. The "IF ... THEN" is used to test a particular condition on execution, at a given time : when the statement is encountered.

"WHEN ... THEN" does also test a condition but the test, rather than being done at a given instant, is performed prior to executing each and every statement of the program. The condition specified in the WHEN clause is continuously monitored until another "WHEN ... THEN" statement is encountered, or until a WHEN request is cancelled by the associated "NEVER WHEN" statement.

Not surprisingly, this statement results in some downgrading, as far as speed is concerned, but its advantages far outweigh this drawback. An example of WHEN usage follows:

Example 1.3

```
125  WHEN Valve_1=0 AND Pressure > 200 THEN Led[3]=1
```

1.2.3.2 Data formatting

Another strength of BASIC-M is that provision is made for formatting both output data and memory-resident data.

1.2.3.2.1 Formatting output data

Before data is transmitted from internal storage to an output device (console, disk, line printer), it goes through an editing process which cannot be easily controlled in standard BASIC. BASIC-M however offers flexible facilities for specifying the format of the data to be output. The PRINT USING statement actually tells the computer to output data contained in its operand list in a format described in its USING clause. BASIC-M provides eight format descriptors that make the language well suited for a wide variety of applications where a versatile formatting of data is at a premium. These descriptors which feature both COBOL and FORTRAN formatting capabilities are fully detailed in chapter 6. The output format can be specified in a literal constant or at execution time; in this latter case, the descriptors are contained in a string variable (see example 1.6).

The three examples below illustrate the type of results that can be achieved.

Example 1.4

```
40  A$="Motorola Semiconductors"
100 PRINT USING "[64,C][/5][X31]",A$,"!"
```

The string "Motorola Semiconductors" is output centered in a 64-column field, followed by 5 empty lines, 31 horizontal spaces and an exclamation mark.

Example 1.5

```
10 DATE = 40579
20 BOOK = 2.19025075E+6
30 PRINT #2 USING 90, DATE, BOOK
90 IMAGE "Date : [C2/2/2][X10]Bookings = [C($)1,3,3(.)2]"
```

The following printout occurs on the line printer :

```
Date : 04/05/79          Bookings = $2,190,250.75
```

Example 1.6

```
10 INPUT Angle, Model$
20 PRINT USING Model$, Angle*PI/180

RUN
? 360
? [1,3,2] radians
6.283E+0 radians
```

1.2.3.2.2 Editing memory-resident data

Most of the format descriptors that apply to the PRINT USING statement are also available for editing memory-resident numeric data, a very valuable feature when working on video RAM. For this purpose, the STR$ built-in function has been enhanced so as to support a second argument which precisely specifies the format descriptors. The STR$(X) function normally converts a numeric value X to a string. A sample program, although not complete, is shown below which causes the string "04/05/79" to be displayed in the top left-hand side corner of a CRT whose video RAM would start at location $4000.

Example 1.7

```
22 DIM Alpha$(16,2) ADDR $4000   \ 16x64 video RAM
24 DATE = 40579
26 Alpha$(1,1) = STR$( DATE, "[C2/2/2]")
```

As in the PRINT USING, the second argument of STR$ which in the above example is a literal constant, may well be a literal variable, thus allowing the user to format data at execution time.

1.2.4 Matrix operations

As already mentioned under 1.2.2.3, BASIC-M supports arrays that can be either one- or two-dimensional. Data items of the same type (byte, integer, real or string) are grouped together to form an array or matrix that can be referred to by a single name. There exists several powerful statements which allow an array to be regarded as a single quantity. This approach results in shorter and faster programs, for it obviates the use of the conventional FOR-NEXT loops operating on every element of the array. The following examples highlight the usefulness of some matrix-oriented statements.

Example 1.8 : initialize a 5x4 matrix A to the value PI

```
10  DIM A(5,4)
20  MAT A = SET [PI]
```

Example 1.9 : a FOR-NEXT loop is used to achieve the same results as in example 1.8

```
10  DIM A(5,4)
20  FOR I = 1 TO 5
30  FOR J = 1 TO 4
40  A(I,J) = PI
```

```
50  NEXT J
60  NEXT I
```

Example 1.10 : input the elements of a 2x2 matrix B from the console (data typed in on the same line).

```
10  DIM B(2,2)
20  MAT INPUT B
```

Example 1.11 : problem definition equivalent to above example (1.10).
A standard BASIC statement is used.

```
10  DIM B(2,2)
20  INPUT B(1,1), B(1,2), B(2,1), B(2,2)
```

Example 1.12 : matrix multiplication

```
10  DIM A(2,4), B(2,3), C(3,4)
20  MAT A = B*C
```

Example 1.13 : matrix inversion

```
10  DIM A(3,3)
20  MAT A = INV(A)
```

1.2.5 Disk I/O

BASIC-M provides versatile statements for exchanging data with a mass-storage media (disk or mini-disk). The following file organizations and access are supported :

1/ sequential organization :
- fixed-length records : sequential or random access.
- variable-length records : sequential access only.

2/ indexed organization :
- fixed-length records : indexed access to a particular record by means of keys.

1.2.6 Built-in functions

BASIC-M includes over 30 intrinsic functions : those commonly found in most BASIC´s , plus several unique ones that considerably ease string processing or mathematical problem solving. Below is a brief list of some advanced functions :

```
ASN(X)          arcsine of X
LOG(X)          natural logarithm of X
DCLOG(X)        decimal logarithm of X
LOC(X)          absolute address of X
SUBSTR(S$,X$)   return position of substring X$ in S$
TRIM$(S$)       strip trailing blanks off S$
```

1.2.7 User-defined functions / procedures

As in the standards, BASIC-M makes provision for user-definition of single-line arithmetic functions, such as the one shown in example 1.14.

Example 1.14

```
10  DEF SURFACE(X) = PI * X^2
20  INPUT "cylinder height and radius ", H, R
30  PRINT "Volume = "; H * SURFACE(R)
```

More interesting is the fact that BASIC-M supports also multi-line user-defined procedures similar to PASCAL´s. Unlike functions, procedures do not return a single value to the calling program; rather they cause the execution of a pre-defined sequence of statements. The sequence is activated by writing the name of the procedure, possibly followed by a list of arguments.

The idea behind user-defined procedures is to improve the overall program structure, thus making it both more readable and secure.

An example of a very simple procedure definition and call follows.

Example 1.15

```
70   GOTO 100
75   DEF DELAY(X)       \ procedure definition
80   FOR BB = 1 TO X    \ software delay
85   NEXT BB
90   RETURN             \ end of procedure definition
95   REM
100  WHEN Pia[3]=1 THEN DELAY(Z)
```

1.2.8 Assembly language interface

In some cases , it might be desirable to perform some time-critical tasks in assembly language. BASIC-M interfaces very easily to user-written assembly language subroutines thereafter referred to as external procedures ; The EXTERNAL statement allows to declare these latter along with their absolute address.

The address of the arguments involved in the external procedures , if any, are passed in a table pointed to by the MC6809 index register.

The external subroutines can also be called as functions returning a value to the calling program.

Below is an example of an external subroutine call and definition; the passing of argument (the address of a string) is illustrated in this sample program.

Example 1.16 : echo an input string by calling the monitor PDATA subroutine

```
15    EXTERNAL ASM ADDR $8000 \ user-written routine
18    INPUT TEXT$             \ read input string
21    TEXT$ = TEXT$ + CHR$(4) \ append terminator
24    CALL ASM( TEXT$ )       \ invoke assembly routine
```

User-written assembly program :

```
      NAM ASM
      ORG $8000
ASM   LDX ,Y       read argument address
      LEAX 1,X     skip string length byte
      JMP  PDATA   call monitor (sub)routine
      END
```

CHAPTER 2

2. PROGRAM ORGANIZATION AND ELEMENTS

2.1 Statement lines

A BASIC-M source program consists of a series of instructions which directs the computer to perform a certain task. Each instruction is called a statement.

BASIC-M has some 30 different kinds of statements and over 30 different built-in functions, which are all discussed separately further on in this manual.

A statement appears wholly on a statement line , thereafter referred to as a "line", which may include up to 80 characters and must be terminated by a carriage return character. No more than one statement can be coded on a line, nor can a statement be continued on the next line.

Each line must be numbered to indicate the normal sequence in which the statements are executed. These line numbers appear at the left end of the line and may be any value from 1 to 65535. Statements may be entered in any order. The computer keeps them in numerical order no matter how they are entered. For example, if statements are input in the sequence 30, 10, 20, the computer arranges them in the order 10, 20, 30.

Good programming practice dictates that the line numbers be separated by some numeric distance, say 10, so that if programming errors are found , or changes made to the program, new lines with numbers in between those which already exist can be inserted.

Upon request, the computer can optionally generate automatic line numbers separated of each other by some user-defined distance. Once the source program has been created, the statements can also be resequenced. The automatic line numbering and resequence system commands are discussed in chapter 13. Here is a brief illustration of their usage.

Example 2.0 : automatic line numbering and resequencing.
(user´s inputs are underlined)

```
READY

N  10 , 2
---------
    10 A=2
       ---
    12 PRINT A
       -------
```

```
    14 STOP
       ----
    16 < CR >    exit program editing by depressing
                 the carriage return key
READY

RESEQ /5
--------

READY

LIST
----

00010  A=2
00015  PRINT A
00020  STOP
```

When a BASIC-M program is executed, execution starts with the first statement in the first line (the statement at the top of the page of a listing); then control flows to the next line down the page (of the program listing). This process continues until a statement is encountered which changes the flow explicitly (i.e, GOTO, GOSUB, NEXT, IF ... etc.), or until a hardware or software event being monitored forces the computer to execute another portion of the source program ; this happens in case of interrupts or of condition monitoring enabled by a WHEN ... THEN statement .

2.2 Statement types

BASIC-M statements may be classified into four basic categories: input/output , arithmetic or string processing, control, and nonexecutable. As for any high-level language , BASIC-M source statements cannot be executed ; the machine-language instructions into which these statements are translated can be executed. But, because "executable statement" is a conventional phrase, it will be used in all subsequent discussions.

2.2.1 Input/output statements

These statements help in exchanging data between the outside world and the user program ; they direct the computer to read or write a record (a collection of data), indicate the device to be used (console, disk, line printer) and may optionally reference a nonexecutable statement which describes the record (IMAGE statement).

Example 2.1 : input/output statement

```
50  INPUT "ENTER PARAMETERS ", A, B, C, X
```

```
70  PRINT USING 90, A*X*X + B*X + C
90  IMAGE "Y = [6,2]"
```

2.2.2 Arithmetic and string processing statements

They constitute the "heart" of most BASIC-M programs, in that they direct the computer to perform certain arithmetic calculations (addition, sine calculation, etc.) or string processing (string concatenation, searching, etc.).

Example 2.2 : arithmetic and string statements

```
10  A$="MOTOROLA "+ B$
20  POSITION = SUBSTR(A$,"RO")
30  Y = SIN(X)
```

2.2.3 Control statements

Normally, statements are executed in the order in which they appear in the source program. Control statements can be used to instruct the computer to change this normal order of execution. For example, control statements can be used to repeat an instruction or series of instructions a specific number of times, or to execute certain instructions only under specified conditions. They can also be used to suspend or terminate program execution.

Example 2.3 : control statements

```
35  FOR I = 1 TO 20
40  TOTAL = TOTAL + A(I)
45  IF TOTAL > MAX THEN PAUSE "OVERFLOW"
50  NEXT I
```

2.2.4 Nonexecutable statements

These are primarily used to give the compiler information it will need to execute other statements. The type declaration statements fall into this category : they govern the allocation of memory for the variables and dictate the type of operations to further occur during program execution (byte or floating-point addition, type conversion, etc.). Examples of other statements of this category include the REM statement (to announce a comment), the IMAGE statement (to specify a printout format), and the DATA statement (to store permanent values in the program).

Example 2.4 : nonexecutable statements

```
10  REM This is a sample program
20  BYTE PIA ADDRESS $8008
30  DATA $FF, 4, 128
40  READ INITLZ
50  PIA = INITLZ
```

statement lines 10, 20 and 30 are examples of nonexecutable statements.

2.3 Statement composition

BASIC-M statements are composed of various combinations of keywords, variable names, constants, expressions, and codes. As many blanks as desired can be inserted between these quantities to improve program readability. However, program editing must obey the following simple rules :

1/ each statement line must not exceed 80 characters.
2/ there must be at least one space between the statement line number and the first element of the statement.
3/ each key word must be followed by a blank.

2.3.1 Character set

BASIC-M programs are written using a subset of the ASCII character set depicted in Appendix A. It is composed of special characters, and of a collection of lower-case, upper-case and numeric characters collectively called alphanumeric characters. The upper-case characters are the characters A through Z ; the lower-case characters are the characters a through z ; the numeric characters are the characters 0 through 9. These are also called the decimal digits. The decimal digits and the characters A through F are collectively called the hexadecimal digits.

The special characters and their meanings or uses, outside of character-string constants and comments, are given in figure 2.1.

	character name	meaning or use
	A-Z , a-z , 0-9	alphanumeric characters
	A-Z	upper-case characters
	a-z	lower-case characters
	0-9	decimal digits
	0-9 , A-F	hexadecimal digits
special characters		
	space	separator, otherwise ignored
$	dollar sign	start of hex constant or string variable terminator
\	backslash	start of comment
[	left bracket	start of bit expression or format descriptor
]	right bracket	end of bit expression or format descriptor
"	double quote	character-string constant delimiter
(	left parenthesis	begin of argument or subscript list
)	right parenthesis	end of argument or subscript list
*	asterisk	multiply
+	plus	add
-	minus	subtract
/	slash	divide or line feed descriptor
^	up-arrow	exponentiation (raise to power)
.	period	decimal point
<	less-than sign	less than or not equal if followed by greater-than sign
>	greater-than sign	greater than or not equal if preceded by less-than sign
#	immediate or pound sign	begin of a logical unit expression or not equal
=	equal sign	equal
,	comma	argument or subscript separator
_	underscore	can be embedded in names to improve readability

Figure 2.1 BASIC-M character set

The general form of a statement line is as follows :

< line number >< blank >< statement body >< comment >< CR >

where :

- the comment field is optional. If any, the comment is preceded by the backslash character and may include any displayable characters.
- CR is the ASCII carriage return character.
- the statement line length must not exceed 80 characters.

The rest of this section identifies the basic elements that may exist in the statement body.

2.3.2 Keywords

Keywords have a special meaning in BASIC-M. They identify operations designated by statements. An alphabetic list of these key words is given in figure 2.2.

ABS	ACS	ADDR	ADDRESS	AND	ASC	ASN
AT	ATN	BYTE	CALL	CHR$	CLOSE	CON
COS	COSH	COTH	DATA	DCLOG	DEF	DIGITS
DIM	END	EOF	ERR	ERROR	EXP	EXT
EXTERNAL	FIRQ	FIX	FKEY	FLOAT	FOR	GO
GOSUB	GOTO	IAND	IDN	IEOR	IMAGE	IND
INPUT	INT	INTEGER	INV	IOR	IRQ	ISHFT
KEY	LEFT$	LEN	LET	LINE	LOC	LOG
MAT	MID$	MOD	NEVER	NEXT	NMI	NOT
ON	OPEN	OR	PAUSE	PEEK	POKE	POS
PRINT	RAN	READ	REM	RESTORE	RETURN	REWIND
RIGHT$	RND	SEQ	SET	SGN	SIN	SINH
SQ	SQR	STEP	STOP	STR$	SUBSTR	TAB
TAN	TANH	THEN	TO	TRIM$	TRN	USING
VAL	WHEN	ZER				

Figure 2.2 BASIC-M keywords

2.3.3 Variable / procedure names

A variable name is a symbolic address selected by the programmer. Although the address remains constant, it is called a variable name because the data contained at the symbolic address may be repeatedly changed during program execution.

A procedure or function name is an identification of a series of statements or instructions which can be executed by specifying in the source program, the name of the procedure / function.

BASIC-M user-defined variable , procedure or function names do not conform with the BASIC standards, in that they can be multi-characters. A name is not limited in length ; BASIC-M stores the names in a symbol table, and each variable, procedure or function is coded internally as a 16-bit pointer to this symbol table ; therefore the user can feel free to use meaningful names without wasting memory space. However, care should be exercised so that the length of any source line is not more than 80 characters. Apart from this restriction, a name must obey the following rules :

- the first character must be an upper-case character.
- a name may consist only of any alphanumeric characters and of the underscore (_), and dollar ($) signs.
- a name must not be identical to any of the BASIC-M reserved words listed in figure 2.2.

legal names

A$, ALPHA$, Alpha$, Alpha_memory, Day_of_the_week_1
E6 , I , PI , Exchange_Rate , Dollar$_value

illegal names

Day_of_the_week:1	name includes a colon
alpha$	name does not begin with an upper-case character
SIN	reserved word. Note that SIN$ would be legal
Alpha memory	embedded space

A variable / procedure / function can be renamed at will, by using the RENAME system command which is discussed in detail in chapter 13. The user must be aware that changing a name does not delete the old name from symbol table. Stated another way, the table expands every time a variable is renamed, thus consuming memory space. Below is an example of the effect of the RENAME command.

Example 2.5 : using the RENAME command

```
LIST

00010  PRINT V1

READY

RENAME V1 Volume_of_cylinder

READY

LIST

00010  PRINT Volume_of_cylinder
```

2.3.4 Constants

Constants , by definition , are unvarying quantities. There are two categories of constants defined in BASIC-M : numeric constants, and literal constants (also called string or character constants).

2.3.4.1 Literal constants

A literal constant is a string of characters enclosed in a pair of double quotation marks. Any letter, digit, or special character can be included in a literal constant. A double quote , however, must be indicated by using two double quotes. For example ,

"""BYE"", HE SAID" represents "BYE", HE SAID

The following are all valid literal constants :

```
" volume of cylinder = "
"3.14"
"%Delta to requirement is above 6"
```

The length of a character constant, when displayed or printed, is the number of characters it contains, including blanks, but excluding the delimiting double quotation marks. Each pair of double quotes used to represent a double quote is counted as one character. The maximum number of characters in a literal constant is limited only by the maximum number of characters on an input line, which is 80.

2.3.4.2 Numeric constants

This category is further subdivided in decimal, and hexadecimal constants.

2.3.4.2.1 Decimal constants

A decimal constant consists of decimal digits with an optional exponent specification. A decimal constant yields a 5-byte data in a format allowing to code quantities in the range 10 raised to power -38 to 10 raised to power +38 , with an accuracy of over 9 digits. A decimal point can be placed anywhere in the digit string. The exponent is specified by writing "E" followed by "+" or "-" or nothing, followed by a digit string for the exponent value itself. The exponent specification, if present , must be preceded either by a decimal digit , or by a decimal point itself immediately preceded by a decimal digit.

valid decimal constants	unvalid decimal constants
3.14159265	3. 14159265
00000300	00000A0
1E12	E12
1.E12	.E12
.314E+00001	.314E/00001

2.3.4.2.2 Hexadecimal constants

BASIC-M can also deal with hexadecimal constants. This is a convenience offered especially for programmers accustomed to machine-language.

A hexadecimal constant consists of a hexadecimal digit string preceded by the dollar sign.
Note that the use of any special character other than the leading dollar sign is prohibited (in particular, the decimal point cannot be used in an hexadecimal constant).
Hexadecimal constants supplied in the source program, are treated as 16-bit signed quantities to represent values in the range -32768 to +32767. However, hexadecimal numbers supplied via an INPUT statements assume the range of decimal constants. The following example illustrates this distinction.

Example 2.6 : hexadecimal constants

```
10  VALUE = $FFFE     !  10  INPUT VALUE
20  PRINT VALUE       !  20  PRINT VALUE
RUN                   !  RUN
-2                    !  ? $FFFE   ( operator´s input )
                      !  65534
```

The hexadecimal constants of the source program may include as many leading zero´s as desired, the sole restriction being that no overflow occurs when the constant is converted to its internal code. Another rule to be observed is that an hexadecimal constant cannot be preceeded by an unary minus. Some more examples are shown below :

valid hex constants	invalid hex constants
$AF	-$AF
$00000000FFE	$0000.FFE
$8008	$8008E+06
$BF74	$BG74

2.3.5 Expressions

An expression is a combination of variable / function names , and constants separated by operators. There are four types of operators : arithmetic , literal , relational , and logical operators. Depending on the variable / function types, and of the operators used in the expression , this latter will be referred to as an aritmetic , or literal, or relational , or logical expression. Expressions are fully discussed in chapter 4. This section simply illustrates the four types of expressions processed in BASIC-M.

2.3.5.1 Arithmetic expressions

Arithmetic expressions can be formed by combining numeric variables / functions and constants (the operands of the expressions) with arithmetic operators. There are five arithmetic operators :

- the "+" operator which implies an addition,
- the "-" operator which implies a substraction,
- the "*" operator which implies a multiplication,
- the "/" operator which implies a division, and
- the "^" operator which implies an exponentiation.

The value of an arithmetic expression is obtained by performing the implied operations on the specified items. For example, if A=4 and B=5, the value of the expression 3*A+B is equal to 3*4+5 , i.e 17 . Note that the constant 3 is the factor by which the variable A only has to be multiplied, and not the quantity A+B . This is due to the fact that the "*" operator has precedence over the "+" operator. The user however, can dictate the flow of calculations by using parentheses. For instance , the expression 3*(A+B) where A and B have the same value as before , will result in the value 27 , because this time , 3 applies to the sum of A and B .

BASIC-M supports mixed-mode expressions ; in other words, the operands involved in arithmetic expressions need not be of the same type. Internal type conversion, resulting type of an expression, and operator precedence are all detailed in chapter 4.

Some examples of arithmetic expressions are :

```
ATN(1) * SQ (R)
PI * R * R               ( spaces are shown for
3.14 * R ^ 2               sake of readability
TOTAL(X,Y) / AMOUNT        only ! )
SIN(X) + DCLOG(Z)
```

2.3.5.2 Literal expressions

A literal expression is a combination of string variables / functions , and/or literal constants, with the concatenation operator "+". The following are valid examples of literal expressions :

```
"GOOD" + "BYE"
 TEXT$ + CHR$(4)
 BUFFER$(3) + "END-OF-LINE"
```

2.3.5.3 Relational expressions

A relational expression compares the value of two arithmetic expressions or two literal expressions. The expressions to be compared are evaluated and then compared according to the definition of the relational operator specified. According to the result, the relational expression is either satisfied (true) or not satisfied (false).

Relational expressions can appear in a BASIC-M program only as part of an IF or WHEN statement.

The relational operators and their definitions are :

"="	equal
"#" or "<>"	not equal
">"	greater than
">="	greater than or equal
"<"	less than
"<="	less than or equal

Below are some examples of statements that involve relational expressions.

```
IF Angle * PI / 180 < 1.57 THEN Rotate(X,Y)
WHEN Pressure = 150 THEN GOSUB 200
IF A$ >= B$ THEN Swap(A$,B$)
```

In the above examples, the relational expressions are those quantities between the key words IF or WHEN , and THEN.

2.3.5.4 Logical expressions

Logical expressions consist of relational expressions combined by logical operators using the ordinary rules of Boolean algebra. For example , the logical expression " A < B AND C # D " is true if the value of A is less than the value of B , and if the value of C is different from the value of D.

The logical operators provided in BASIC-M are :

"NOT"	logical expression is true if relational expression is false, and vice versa.
"AND"	logical expression is true if both relational expressions are true.
"OR"	logical expression is true if either relational expression is true.

Logical expressions can only appear in the IF or WHEN statements, as presented in the following examples :

```
WHEN NOT(Pressure < 150) OR Temp >= 273 then Alarm
IF A$ < Company_name$  AND  A=3  OR  B#7 THEN  100
```

Logical operators hierarchy is also discussed in chapter 4.

2.3.6 Codes

The BASIC-M language includes several codes, that are not executed but rather gives the computer information he needs at execution time. These codes are supplied by reserved words or special characters. Their main function is to describe a printout format, a disk file organization, or a data type. For instance , in the following :

```
10  PRINT USING "[/3]" , A
20  OPEN #98, "STOCK" , I , RAN
30  INTEGER SCRATCH, TEMP
```

The slash character followed by 3 in statement 10 , tells the computer to output 3 line feed characters to the console prior to printing variable A . Similarly, in statement 20 , I specifies that the file named "STOCK" is to be opened for input , while the reserved word "RAN" implies a random access.

This chapter was intended to describe the organization of a BASIC-M program , and the main elements which compose the various statements. The next chapter goes into more detail as to the data types and structures defined in the language.

CHAPTER 3

3. DATA TYPES AND STRUCTURES

When a language is evaluated, not only does one have to look at its statement repertory. The statements just show the actions that may be taken. Equally important are the data which can be operated upon by the statements. A data type determines the set of values which variables and functions of that type may assume.

BASIC-M includes the standard BASIC data types (real and character), plus two non-standard types (byte and integer) which are frequently used in a microprocessor environment. This chapter discusses important subjects related to data types, such as internal representation , magnitude of the data, accuracy , type declaration or other arrangement of data in structures. These concepts are especially important to those people wishing to interface a BASIC-M program to assembly-language subroutines, in that it describes the format of the argument data.

3.1 Standard BASIC data types

3.1.1 Character data

BASIC-M adheres to the standard convention that any variable name ending with a dollar sign ($) defines the data it represents as a character, or string data. In the BASIC-M language, this convention also extends to the user-defined functions , either internal or external, and to the library functions.

The following statements all define character variables or functions.

Example 3.0 : variables or functions assuming the character type.

```
10  EXTERNAL HEADing$ ADDRESS 1024
20  A$ = STR$(A)
30  DEF Catenate$(X$,Y) =  X$ + CHR$(Y)
40  TEXT$ = Catenate$(A$,4)
50  HEADing$(TEXT$)
```

A character data consists of 31 ASCII characters, and is internally coded on 32 consecutive bytes in the following format :

L A A A A A .... A A

where :

A stands for an ASCII character.
L is the string length byte, which holds the current length of the string (0 to 31).

Character data are left-justified, and the non-significant ASCII bytes, if any, are filled with blanks. Example 3.1 illustrates the successive internal coding of a string variable in the course of program execution.

```
Example 3.1 : internal representation of
-----------   a string variable

                variable    L A A A . . . . . . . ... A

10  A$="MC68"    (A$)       4 M C 6 8 b b b b b ... b
20  B$="09"      (B$)       2 0 9 b b b b b b b ... b
30  A$=A$+B$     (A$)       6 M C 6 8 0 9 b b b ... b
                            ( b stands for blank )
```

3.1.2 Real data

The real type is the standard numeric data type. Any variable name which does not end with a dollar sign and which has not been explicitly declared via a type declaration statement, defines the data it represents as a real quantity. This standard convention also applies, in BASIC-M, to internal or external user-defined functions.

Some examples of real variables or functions are shown below.

```
Example 3.2 : Real type variables
-----------   or functions.

10  DEF HORIZ(R,TETA) = R*COS(TETA)
20  EXTERNAL PLOT(X,Y) ADDR $E000
30  Z = HORIZ(Radius,30)
40  PLOT(Z,W)
```

A real data is stored internally on 5 consecutive bytes, in a floating-point format that permits representation of null, positive, and negative numbers in the dynamic range E-38 to E+38 , with an accuracy of 9 digits.

The internal representation is as follows :

Byte 0 : binary two´s complement exponent

Byte 1-4 : 31-bit, normalized, binary mantissa in 2´s complement form. least-significant-bit of byte 4 stores sign.

The decimal point lies between the exponent (byte 0), and the most-significant bit of the mantissa (byte 1).

The mantissa is in the range 0 included to 1 excluded.

In this format, a number is considered as normalized if the most significant bit of its mantissa (byte 1 - leftmost bit) is the complement of the sign bit , the exception being the value 0 which is coded as all zero´s. Example 3.3 presents a few internal representations of real data.

Example 3.3 : coding real numbers

```
             byte 0   byte 1   byte 2   byte 3   byte 4
            ( expo.)                                   Sign
                                                        v
      .75 = 00000000 11000000 00000000 00000000 00000000
     -.75 = 00000000 01000000 00000000 00000000 00000001
       24 = 00000101 11000000 00000000 00000000 00000000 (.75x32)
      -24 = 00000101 01000000 00000000 00000000 00000001
3.14159265= 00000010 11001001 00001111 11011010 10101110
```

Example 3.15 shows how a simple BASIC-M program can be written to find out the internal representation of real values.

3.2 Non-standard BASIC data types

3.2.1 Byte data

A BASIC-M variable (but not a user-defined function!) may consist of one byte only. Byte data are 8-bit unsigned quantities in the range 0 to 255. Should data be outside this range at execution time, a runtime error will occur.

Byte variables must be explicitly declared by the BYTE statement which applies to all the variables of the input line. As will be seen later, the absolute address of a byte variable can be specified via the ADDRESS or ADDR key words.

Byte variable names must not end with a dollar sign ($) which implicitly defines string quantities (see 3.1.1).

An example of byte type declaration and usage is presented next.

Example 3.4 : Byte variables

```
40   BYTE CRTC ADDRESS $EF00, Scratch, Temp
50   BYTE SSDA ADDR $EF04
```

```
60  SSDA = 4               \ initialize SSDA
70  CRTC = $12             \ initialize CRTC
80  SSDA = SSDA + 252      \ faulty !!!
```

Statement 80 will yield a runtime error message since the resulting value of SSDA would be 256, which is outside the allowed range.

3.2.2 Integer data

Integer variables (not user-defined functions !) are 16-bit binary signed data in 2's complement form, internally stored on 2 consecutive bytes. These variables hold data in the range -32768 to +32767.

The sign bit is the most significant bit of the 16-bit data (first byte, bit #7 , visually leftmost bit).

Integer variables must be explicitly declared via the INTEGER statement which is syntaxically similar to the BYTE statement.

Integer variable names must not end with a dollar sign.

The following statements define and use integer variables.

Example 3.5 : integer variables

```
2  INTEGER IRQ_Vector ADDRESS $A000
4  INTEGER A, B, C
6  C = A + B            \ integer add
```

3.3 Access to individual bits

A very handy feature of BASIC-M is that the user may address each and every bit of a non-subscripted byte/integer variable, just by specifying its position within the variable. This greatly eases solving of process-control applications.

The position of the bit which is to be set, cleared or tested, within the variable is specified in a pair of brackets, as shown in the general form :

VAR[exp]

where :

VAR is a byte or integer variable name , and
exp is an arithmetic expression that indicates the number of the bit to be accessed.
Note that the most significant bit of a byte, respectively

integer variable corresponds to the number 7, respectively 15, the least significant bit being in both cases bit #0. Example 3.6 indicates what can be done with bit addressing.

Example 3.6 : bit access

problem : two switches are connected to two PIA lines, PA0 and PA1. Switch on the led drived from PIA line PA7, if both switches are not in the same position. Perform a continuous monitoring.
(PIA initialization is not shown)

```
10  BYTE PIA ADDRESS $7004
15  PIA[7] = 0        \ default to led off
20  WHEN PIA[0] # PIA[1] THEN PIA[7] = 1
```

Example 3.7 : count the number of bits set in the integer variable DEMO

```
30  INTEGER DEMO
32  COUNT = 0         \ set up counter
34  FOR I = 0 TO 15
36  COUNT = COUNT + DEMO[I]
38  NEXT I            \ study next bit
```

This example can be applied to byte variables, for computing the parity of an ASCII character

3.4 Mixing data types in expressions

BASIC-M supports mixed-mode expressions, that is, the elements of expressions can differ as to their type. This holds true also for an assignment statement : both side of the assignment operator ("=" sign) need not be of the same type. However, avoiding mixed types will have a significant impact on the overall program compactness and speed, since the compiler will not generate the necessary calls to the conversion routines. Optimizing compiler output is a subject covered in chapter 13 ; to give the reader an idea of the improvement that can be reached , as far as object program size and execution speed are concerned, let´s consider the figures shown in example 3.8

Example 3.8 : optimizing program size and speed.

The two BASIC-M programs shown below produce the same result

```
10  INTEGER A        !  10  INTEGER A, B, C, D
20  BYTE B           !  20  A = B + C - D
30  A = B + C - D    !
                     !
size  : 37 bytes     !  size  : 34 bytes
speed : 1554 cycles  !  speed : 183 cycles
```

The example just presented must not be used as a basis for conclusion, the speed being dependent on the current value of variables. As a general rule programs that do not use mixed-mode expressions will produce better results.

3.5 Data structure : the array

3.5.1 General description

An array is a collection of data elements of the same data type (character, real, byte, or integer) , that can be referred to by a single name. Arrays can be either one- or two-dimensional. A one-dimensional array is also called a vector ; a two-dimensional array is often referred to as a matrix. A one-dimensional array can be thought of as a row of successive data items. A two-dimensional array can be seen as a matrix of rows and columns. Figure 3.1 shows a schematic representation of both types of arrays.

<u>One-dimensional array named A</u>

```
A(1)          A(2)          A(3)          A(4)
```

<u>Two-dimensional array named B</u>

```
B(1,1)        B(1,2)
B(2,1)        B(2,2)
B(3,1)        B(3,2)
```

<u>Figure 3.1 Schematic representation of arrays</u>

Each element in an array is referred to by the name of the array followed by a subscript in parentheses, which indicates the position of the element within the array. The general form for referring to an array element is :

Array name (rowexp , colexp)

where :

- Array name is the name of the entire array,

- rowexp and colexp are positive arithmetic expressions whose truncated integer values are greater than zero and less than or equal to the corresponding dimension of the array.

The dimensions of an array and the number of elements in each dimension are established either implicitly, or explicitly if the array is declared via DIM, BYTE, or INTEGER statements.

An array A(N,M) spans NxMxX+2 bytes in data section (RAM) , and 8 bytes in program section (ROM), where X stands for the elementary data length in bytes. Thus, the integer array I(5) occupies 2+5x2=12 bytes of RAM, and the character array S$(3,4) , 2+3x4x32=386 bytes.

3.5.2 Array declaration

3.5.2.1 Declaring byte and integer arrays

Byte and integer arrays must be explicitly declared by using the BYTE and INTEGER statements respectively, whose general form is as follows :

```
BYTE    Arr1(x1,y1), Arr2(x2,y2), ... , ArrN(xN,yN)
INTEGER Arr1(x1,y1), Arr2(x2,y2), ... , ArrN(xN,yN)
```

where :

- ArrI are variable names that must not end with a dollar sign.
- each pair xI,yI defines the maximum number of elements per row and per column respectively (yI is omitted for one-dimensional arrays).
- the BYTE or INTEGER key words apply to all the arrays of the statement line.
- xI must be in the range 1 to 65535 for one-dimensional arrays.
- each xI, yI, must be in the range 1 to 255 for two-dimensional arrays.
- the number of dimensions must not exceed 2.

The following are examples of such declarations.

Example 3.9 : declaration of byte and integer arrays

valid declarations :

```
10  BYTE PIA(2), Test_Memory(65535), CRTC(18)
20  INTEGER WORD(16,16), Temp(20)
```

invalid declarations :

```
10  BYTE Cube(2,2,2)   \ 3 dimensions
20  INTEGER Mem(0,16) \ 1st dimension is null
30  BYTE A(10,256)     \ 2nd dimension > 255
```

3.5.2.2 Declaring real and character arrays

Real and character arrays can be declared either explicitly by use of the DIM statement or implicitly by a reference to an element of an array that has not been explicitly declared.

The DIM statement is syntactically identical to the BYTE or INTEGER statements described under 3.5.2.1, but the following rules apply :

- an array name ending with a dollar sign defines a character array.
- the maximum number of dimensions is two.
- one-dimensional arrays may contain up to 65535 (this far exceeds the system memory !)
- two-dimensional arrays may contain up to 255 elements per row and per column.

Some examples are shown below.

Example 3.10 : declaring explicitly real and character arrays.

```
2  DIM A$(2,3), TTT(128), LINE$(2)
4  DIM TEXT$(60,2), Float_mat(4,8)
```

array	type	# of elements	RAM size
A$	character	2x3=6	6x32+2=194
TTT	real	128	128x5+2=642
LINE$	character	2	2x32+2=66
TEXT$	character	2x60=120	120x32+2=3842
Float_mat	real	4x8=32	32x5+2=162

When an array is declared implicitly, by a reference to one of its elements when its name has not appeared in a prior DIM statement, it will have the number of dimensions specified in the reference, and each dimension will contain :

10 elements for real arrays,
5 elements for character arrays.

For example, if no prior DIM statements exists for arrays named BUFF$ and Values, the statements :

```
85  IF Values(3) > 3 * AVerage THEN GOSUB 300
          :
300 PRINT BUFF$(I,4)
```

will establish a one-dimensional real array "Values" containing 10 elements, i.e 2+5x10=52 bytes, and a two-dimensional character array "BUFF$" containing 5x5 elements, i.e 2+25x32=802 bytes !. Omitting array declarations may result in a workspace buffer overflow that would preclude program compilation and execution. Therefore, the user is strongly recommended to explicitly declare the arrays.

3.5.3 Arrangement of arrays in storage

One-dimensional arrays are stored in ascending storage locations. Thus, the array A(8) is arranged in the order :

A(1) A(2) A(3) A(4) A(5) A(6) A(7) A(8)

Two-dimensional arrays are stored in ascending storage locations, with the value of the second of the subscript quantities increasing most rapidly, and the value of the first increasing least rapidly. Stated another way, arrays are stored row by row. For example, the array B(3,2) presented in 3.5.1, is arranged in storage as follows :

B(1,1) B(1,2) B(2,1) B(2,2) B(3,1) B(3,2)

This approach contrasts with the way arrays are stored in FORTRAN (column by column). However, this arrangement is definitely more consistent and practical when working on video RAM´s.

3.5.4 Matrix-oriented statements

BASIC-M includes a set of statements that allow the user to handle arrays considered as single entities, thus eliminating the need for operating on the individual elements of the arrays. Chapter 9 covers entirely this powerful set of statements. The following example is one among many, that is presented here just to introduce the concept of matrix re-sizing.

Example 3.11 : Matrix assignment
Result : copy the elements of matrix B(2,3) into the matrix A(4,6)

```
10  DIM A(4,6)  , B(2,3)
20  MAT A = B
```

3.5.5 Matrix re-sizing

Some matrix-oriented statements dynamically alter the size of the arrays. In example 3.11 for instance, the assignment of matrix B to matrix A causes this latter to be re-sized to the size of B, i.e 2x3. The size of a matrix can only be decreased from its original size. In other words ,

the assignment "MAT B = A" would be illegal.
This feature is of benefit in several situations, like in solving linear systems of equations by using matrix inversion.

3.6 Variable and array address control. Equivalence.

Variables and arrays are normally assigned storage locations which are not under user control.

The ADDRESS or ADDR key words provides an easy means to specify their starting address, and also to achieve the effect of the FORTRAN EQUIVALENCE statement. The following are some examples with explanations of what can be done.

Example 3.12 : Initialize a PIA

```
10  BYTE PIA(2) ADDRESS $8008
20  PIA(1) = 1   \ Data Direction Reg set-up
30  PIA(2) = 4   \ Control  Register  set-up
```

Alternate solution :

```
10  BYTE PIA(2) ADDRESS $8008
20  INTEGER PIA_EQU ADDRESS PIA \ equivalence
30  PIA_EQU = $104  \ same effect as lines 20
                      and 30 of previous example
```

Example 3.13 : display an input string in the first line of a video RAM based at hex address $E000

```
10  DIM Alpha_mem$(16,2) ADDR $E000
15  BYTE DUMMY ADDRESS $E000
20  INPUT Alpha_Mem$(1,1)
25  DUMMY = $20  \ overwrite string length byte
                ( see 3.1.1 ) with a blank
```

Example 3.14 : erase video RAM (fill it with blanks)

```
10  DIM Alpha_mem$(16,2) ADDR $E000
20  BYTE Charac(1024) ADDR Alpha_mem$
30  MAT  Charac = SET [32]
```

alternate solution :

```
20  INTEGER Word(512) ADDR Alpha_mem$
30  MAT Word = SET [$2020]
```

Example 3.15 : print the binary internal form of a real number.

```
10   INPUT A
20   BYTE OVL(5) ADDR A, CURRENT
30   FOR I=1 TO 5
40   CURRENT = OVL(I)
50   FOR J=7 TO 0 STEP -1
60   PRINT CURRENT[J] ; \ print bit
70   NEXT J
80   PRINT
90   NEXT I             \ next byte
95   GOTO 10

RUN
? -3
0 0 0 0 0 0 1 0    byte 0 - exponent
0 1 0 0 0 0 0 0    byte 1 - mantissa
0 0 0 0 0 0 0 0    byte 2 -     "
0 0 0 0 0 0 0 0    byte 3 -     "
0 0 0 0 0 0 0 1    byte 4 - mantissa + sign
```

CHAPTER 4

4.0 EXPRESSIONS

This chapter gives additional information on the expressions which were already introduced in paragraph 2.3.5.

4.1 Rules for writing arithmetic expressions

The following rules must be followed when writing arithmetic expressions which contain two or more constants and/or variables (for the sake of simplicity, functions will be considered as variables throughout this chapter).

4.1.1 Separation of constants and variables

In regular mathemetic notation, to indicate "A times B", it is common to write "AB". In BASIC-M, this would refer to a variable named "AB". In effect, each constant and/or variable must be separated by an operator to explicitly indicate the desired computation. Therefore to indicate "A times B", one should write "A*B".

Arithmetic operators all require two operands. However the "+" and "-" signs can also be used as positive / negative operators in two situations :

- following a left parenthesis and preceding an arithmetic expression.
- as the leftmost character in an entire arithmetic expression that is not preceded by an operator.

For example :

```
-A+(-B) and B-(-C) are valid
A+-B    and B--2   are invalid
```

4.1.2 Separation of arithmetic operators

Two or more arithmetic operators can never appear in sequence in an arithmetic expression. For example , to indicate "A times the negative value -3", the expression cannot be written as "A*-3" but should be written as "A*(-3)". This illustrates how parentheses can be used to separate arithmetic operators. Other uses of parentheses are discussed next.
Note that the FORTRAN operator "**", expressing an exponentiation, is not supported in BASIC-M, nor does the language support bitwise operators. Logical AND, inclusive

OR, exclusive OR, and shift operations are all performed by using built-in functions.

4.2 Order of evaluation of arithmetic expressions

The order of computation of arithmetic expressions is based on the hierarchy (or precedence) of the operators involved. Evaluation is performed from left to right according to the hierarchy shown below.

hierarchy	Operation
1st	parenthetical expressions
2nd	unary minus
3rd	exponentiation
4th	multiplication and division
5th	addition and subtraction

In other words, the hierarchy goes from what might be considered the most difficult to the least difficult, as is illustrated in example 4.0.

Example 4.0 : operator hierarchy.

Assuming A, B, and C have been assigned the values -1, 9, and 3 respectively, the computation of the expression -A^2+B/C*4 is performed as follows :

```
- A ^ 2 + B / C * 4        1 ^ 2 + 9 / 3 * 4
                             1   + 9 / 3 * 4
                             1   +   3   * 4
                             1   +     12
                                 13
```

Parentheses may be used to dictate the order in which calculations are to be performed and to alter the lower three levels of hierarchy. For example, suppose you desire to add A to B and then triple the sum. Instead of writing :

```
      X = A + B
      Z = X * 3

or :  Z = A * 3 + B * 3
```

you could write

```
Z = ( A + B ) * 3
```

Note that the use of parentheses reverses the normal levels of hierarchy since the computer will first add and then multiply. If the programmer has any doubt as to the order of evaluation of expressions, it is suggested that parentheses be used. Unnecessary parentheses do not affect at all the execution time of a program.

4.3 Mixed-mode arithmetic expressions Impact on program size and speed

As already mentioned in paragraph 3.4, mixed-mode expressions are allowed in BASIC-M. In normal cases, the programmer does not have to bother about the types of the variables and/or constants of the expressions. This is especially true when running a program that meets the BASIC standards. However, BASIC-M makes provision for unique data types discussed in chapter 3. Using these types can result in drastic improvements of the object code size and execution speed, as reflected in the sample program below.

Example 4.1 : object code size and speed improvement by avoiding mixed-mode expressions.

The two programs shown below produce the same result.

Program A

```
 5  BYTE Y
10  DIM B(5,5)
20  A=B(3*X+Y,Z/W)
```

Program B

```
10  INTEGER A,B(5,5),X,Y,Z,W
20  A=B($3*X+Y,Z/W)
```

$$\text{code size} \; \frac{\text{Prog B}}{\text{Prog A}} = 0.87 \qquad \text{exec. time} \; \frac{\text{Prog B}}{\text{Prog A}} = 0.34$$

These improvements are due to the fact that variables appearing in program B are all defined as being of the same type. Likewise, the constant 3, written as an hexadecimal constant $3, defines an integer constant. Therefore the elements of the expression of line 20 are all integers, and the compiler will not generate any call to type conversion routines. In addition to that, the expressions representing subscript quantities are already of the integer type, which is the type expected by the compiler. Should the result of the expressions be of the real type, the compiler would have generated a call to a real-to-integer conversion routine, resulting in a larger code size and lower execution speed. The following is the code generated by the compiler for addressing an element A(X+Y) of a vector A, given two situations.

Example 4.2 : code generation for addressing the element A(X+Y) of a vector

1/ X REAL, Y BYTE	2/ X,Y INTEGERS
LDX #X	LDX #X
JSR LF	JSR LI
LDX #Y	LDX #Y
JSR LB	JSR LI
JSR BTOF	JSR IADD
JSR FADD	LEAX A,PCR
JSR FTOI	JSR INDEX
LEAX A,PCR	
JSR INDEX	

LF , LB, and LI are runtime routines that load onto the MC6809 User Stack a real, byte and integer data respectively addressed by the X-register.
BTOF and FTOI are runtime conversion routines to convert stacked data from byte-to-real and real-to-integer respectively.
FADD and IADD perform the addition of two real and integer data respectively on the User Stack.
INDEX returns in the X-register the address of the element in matrix A whose subscript is the 16-bit last stacked data. The information that pertains to matrix A (number of dimensions, size of dimensions, element size, absolute start address, and dynamic size word address) is stored in the program section (rom-able object code section), hence the use of the program counter relative addressing mode to access it in a fashion that preserves position-independence.

This section describes the benefits of avoiding mixed-mode expressions, and of using only the compiler default types (for instance, subscripts default to the integer type). The next paragraph discusses the data type of the results of arithmetic expressions. Because expressions may include built-in functions (SIN, SQR, MOD, ... etc.) , it is also worthwhile to know their data types, along with the default type of the arguments. This information is given in chapter 13.

4.4 Data types produced by arithmetic expressions

The type of the result of an arithmetic operation depends upon the type of the two operands (primary) involved in the operation. The table below gives the correspondence between the type of the result and the type of the primary, assuming an expression of the form :

Prim1 oper Prim2

where Priml, Prim2 represent the the types of the two primaries (variables or functions), and oper is either one of the operators : +,-,*,/,^.

Priml \ Prim2	byte	integer	real
byte	byte	integer	real
integer	integer	integer	real
real	real	real	real

NOTES :

Numeric constants are always real, except if preceded by a dollar sign which implies an hexadecimal constant further regarded as an integer.

At least one of the primaries involved in an exponentiation must be of the real type. The result type of an exponentiation is then always real. A positive value can be raised to any value, whereas a negative value can only be raised to an integer number (one whose fractionary part is null).

When division is applied to two bytes or integers, the answer is truncated and a byte or integer result respectively is produced. For example, if A=5, B=2, and if A and B are bytes or integers , then the expression A/B would yield a result of 2.

4.5 Literal expressions

As defined earlier in chapter 3, literal expressions are those quantities containing two or more of the following elements :

- character variables,
- character constants,
- character user-defined functions, and/or
- character built-in functions,

separated by the concatenation operator "+".

All the character elements mentioned above contain a maximum of 31 ASCII characters, except the character constants which are only limited in size by the input line length (80 characters). Example 4.3 illustrates some uses of literal expressions.

Example 4.3 : literal expressions

```
10  A$="H"
20  B$="E"
30  C$="L"
40  D$=CHR$(79)    \ ASCII "O"
50  IF INPUT$ < REF$ THEN 80
60  PRINT "GO TO ";A$+B$+C$+C$
70  GOTO 320
80  PRINT A$+B$+C$+C$+D$
90  GOTO 300
```

When character data appears in a relational expression (as is the case in line 50 of the previous example), it is evaluated according to the sequence of the ASCII codes (see Appendix A), character by character, from left to right. Thus , the following relational expressions would all be satisfied :

```
"ABC" = "ABC"
"ABCDEF" < "ABCEEF"
"ABcd" > "ABCD"
"AB" > "12"
```

When character operands of different length are compared, the shorter operand is considered to be extended on the right with blanks to the length of the longer operands. Thus, when comparing "AB " to "ABC", one effectively compares the strings "ABb" and "ABC", where b stands for the blank character. Likewise, "ABC "="ABC".

4.6 Evaluation of logical expressions

Unless you change the order in which logical operators are applied to their logical operands by using parentheses, high-precedence logical operators are applied before low-precedence logical operators, and equal-precedence operators are applied from left to right. The precedence of the logical operators is shown in the following table.

Logical operator	Precedence
NOT	high
AND	low
OR	low

The logical expression :
NOT(A < B) OR C # D , is true if
A is greater than or equal to B, or if
C is different from D.

The logical expression :
C = D OR E = F AND G = H , is true

either if the values of C and D are equal, or if the values of E and F are equal and the values of G and H are equal.

The logical expression :
(C = D OR E = F) AND G = H , is true if G equal H, and if either C equals D or E equals F.

CHAPTER 5

5. BASIC-M SIMPLE STATEMENTS

This chapter discusses the statements which are most frequently used in a BASIC program. More advanced statements, those unique to BASIC-M, are described in subsequent chapters of the manual.

5.1 The LET statement

General form : LET variable = expression

Purpose : used to assign or specify the value of a variable.

Comments : the assignment operator "=" is read "takes the value of", rather than "equals". Therefore, it is possible to write :

```
10 LET I = I + 1
```

which is interpreted as : "LET I take the value of (the current value of) I , plus 1".

The variable and the expression on the right-hand side of the "=" must represent both numeric quantities, or both character quantities. In the former case, the variable and the expression need not be of the same numeric data type. Below are three examples of the LET statement :

```
15 LET PI = 3.14
20 LET Message$ = "HELLO" + CHR$(4)
25 LET A = B$ + C$ \ INVALID EXAMPLE
```

The word "LET" can be omitted in an assignement statement. These two statements :

```
30 LET A = B + C
35 A = B + C
```

mean exactly the same.

5.2 The REM statement

General form : REM any series of characters

Purpose : Allows insertion of a line of comment in the listing of a program.

Comments : REM lines are ignored when the program is executing. They are solely for information and make your programs easier to work with, and easier for other people to use.

As discussed under 2.3.1, an alternate method to insert comments is to append them at the end of a statement line, right after the backslash character.

Example 5.0 :
```
50 REM Initialize variables
60 LET A = 1.414 \ Square root of 2
```

5.3 The READ, DATA, and RESTORE statements

The assignment statement is often used to initialize variables to specific values. The DATA and its associated READ and RESTORE statements provide an alternate solution for initializing variables, and result in fewer lines of program.

General form :
```
DATA cst1, cst2, ..., cstN
READ var1, var2, ..., varN
RESTORE
```

where :

cst1, cst2, cstN denote numeric or string constants, and var1, var2, varN identify numeric or string variables.

Purpose : The DATA statement causes values to be placed in an internal data table, sequentially, i.e in the order in which they are entered. There may be several DATA statements in a program. For instance, the following :

```
20 DATA 3.14, $AA, "HELLO", 22, -6.28
```

is equivalent to the set of lines :

```
12 DATA 3.14, $AA
14 DATA "HELLO",22, -6.28
```

Once the values are in the data table, they can be assigned to variables by using the READ statement, as illustrated below :

```
30 READ PI, PATTERN, MESG$, N, REF
```

Note that the previous line is functionally equivalent to the following set of assignments :

```
25 LET PI = 3.14
30 LET PATTERN = $AA
35 LET MESG$ = "HELLO"
40 LET N = 22
45 LET REF = -6.28
```

The READ statement locates the values in the data table sequentially and assigns them, in order, to the variables. This can be done via several READ statements, not necessarily a single one. For example, the READ statement shown before can be split in two, as illustrated next :

```
100 READ PI, PATTERN
```

would cause the first two constants in the table to be assigned to PI and PATTERN respectively. Another READ statement would take up where the last one left off. Thus :

```
110 READ MESG$, N, REF
```

would complete the assignment.

If you wish, you can use the values in the data table more than once. At any point in a program, one can instruct that values be assigned from the beginning of the table again, even all the values in the table have not been read. The RESTORE statement is used to point back to the beginning of the table.

Comments :

. Care must be exercised so as not to read more values than the table contains. This would result in a runtime error.
. DATA statements need not be grouped together: any statement line can be placed between two DATA statements.
. DATA statement(s) can be located anywhere, even after READ statement(s).
. A number is converted to a string (free format) if assigned to a character variable.
. If a string value is assigned to a numeric variable an attempt is made to convert it prior to assignment (see STR$ and VAL built-in functions).

Example 5.1 :

```
10 DATA 1,2,3,4,5,6
20 READ A,B,C \ A=1, B=2, C=3
30 READ D  \ D=4
40 RESTORE
50 READ G,H \ G=1, H=2
```

5.4 The console INPUT statement

General form : INPUT var1, var2, ..., varN

Purpose : assigns values input from the console to variables. Unlike the LET and READ statement which both allow the supply of constant values to variables (values that must be known when the program is prepared), INPUT waits until the program is running before actually supplying these values.

Comments : When an INPUT statement is encountered, the program comes to a halt, and a question mark is output to the console. The program does not continue execution until the input values have been all entered and assigned to the variables listed in the INPUT statement. Each line of input data must be terminated by a carriage return, and the data must be separated by a comma or a space. A question mark is printed on the console when more values are required to satisfy an input statement.

Should more values be entered than implied by the INPUT statement , extra ones are merely ignored.

Control codes :The following three control codes are provided for editing an input data line :

RUBOUT deletes the last entered character
CTRL-H has the same effect as RUBOUT
CTRL-X deletes the whole input line.

Break : As mentioned before, the ? character continues to be printed after each response until enough numbers are typed in. Should the user desire to abort the INPUT statement , he may enter an exclamation mark (!). This causes the variables, that were not yet supplied with values by the INPUT process, to stay at their current values. For example , given the statement :

```
10 INPUT A,B,C,D
```

and the following input lines :

```
? 22 $44 (carriage return)
? !  (! is typed in by operator)
```

the values 22 and $44 will be assigned to A and B respectively, whereas the current values of C and D will remain unchanged.

Entering strings : The previous example shows an input line where the input data are separated by a space character. Neither the space nor the comma however can be used as a delimiter when entering character data, since these two characters can be embedded in a literal. If several literal constants are to be entered on the same input line, one must use delimiting quotes. Thus, given the statement :

```
100 INPUT A$,B$
```

If the input line looks like :

```
? THIS IS
```

the string "THIS IS" will be assigned to A$, whereas :

```
? "THIS" "IS"
```

will assign "THIS" to A$, and "IS" to B$.

If quotes are not used to delimit the string values, the first 31 characters typed in are put into the first string variable, the next 31 characters in the second variable, ..., etc; the last variable assigned is truncated to the remaining number of characters in the line.

Error checking : The INPUT statement checks for valid data. Should an erroneous data be entered, an error message is printed and the operator is requested to re-enter the values from where the error was detected. For instance, if as a response to the statement :

```
50 INPUT A, B, C$
```

one enters the following :

```
? 3 "HELLO" "enter"
    ^
RETYPE FROM ARROW
?
```

the value 3 assigned to A is preserved, whereas one must re-enter the values for B (that was erroneously assigned a string value), and for C$.

Input prompt : Since a BASIC program can contain several INPUT statements, one has to keep track of which one is being executed. This may be solved by preceeding the INPUT statement with a PRINT statement, or better, by specifying an input prompt in the INPUT statement itself. The prompt will be automatically displayed when the associated INPUT statement becomes active. The input prompt is specified as a literal constant that follows the INPUT key word, as illustrated below :

```
10 INPUT "enter coordinates ", X, Y

RUN
enter coordinates ? 312 256
```

5.5 The PRINT statement - simple form.

This paragraph discusses the simplest form of the PRINT statement, the one used to output data in free format to the system console or line printer. Formatting data via the PRINT USING statement, and saving data onto a diskette file are described in separate chapters.

General form : PRINT #LU expl dell ... expN delN

where :

- LU is an expression yielding a logical unit number in the range 0 to 255. If LU=0 or LU=1, the printout occurs on the console ; if LU=2, the printout is directed to the system line printer; if LU > 2 , then the operands are saved onto a diskette file. The logical unit specification is optional (#LU). If not present, the printout is directed to the system console.

- expl, ..., expN are arithmetic or literal expressions whose values are printed to the logical unit.

- dell, ..., delN are PRINT delimiters, that affect how spacing is to be performed between the printed values. Either one of the characters "," or ";" can be used as delimiters. In addition, the last delimiter delN can be omitted.

Purpose : The PRINT statement causes the values yielded by the expressions expl, ..., expN, to be converted to a readable form and printed onto the system console or line printer. The values are printed in the order in which they appear in the operand list.

String printout : Character values are printed as the ASCII equivalent of the string contents, i.e., "STRING" is printed as STRING.

Numeric printout : Numeric values are printed in either of the following forms :

1/ sign d.ddddddddE esgn xx

if the magnitude of the number is greater than 2 raised to power 29 (appr. 5.3687E+08), or if the magnitude of the number is less than 2 raised to power -4 (.0625).
The digit d to the left of the decimal point is always different from zero ; trailing zeros are suppressed ; esgn represents the exponent sign ; xx represents a two-digit exponent. The printed form corresponds to a value of (sign d.dddddddd) * 10 raised to power (esgn xx).

2/ sign ddddddddd

if the value is an integer whose magnitude is less than 2 raised to power 29 (appr. 5.3687E+8) , sign is a "-" sign if the number is negative (otherwise this position is omitted), and ddddddddd are digits with leading zeros stripped off printout.

3/ sign dddd.ddddd

if none of the previous conditions describe the value; a maximum of 9 digits are printed in this form; "." represents the decimal point and may be anywhere within the digit string ; leading and trailing zeros to the right of the decimal point are suppressed.

Delimiters : Field separators may be either "," or ";" . The comma causes tabbing between printed fields; it forces the terminal to space to the column such that the column number modulo 20 is one ; stated another way , there are column boundaries at 21,41,61, ..., etc.
The semicolon causes a space to be printed if the value to its left is numeric ; if the value to its left is a string , the semicolon prints nothing. Below are some examples of printout :

Example 5.2 : Effect of PRINT delimiters.

```
10  Pi=3.14
20  S$="good"
30  T$="bye"
40  PRINT Pi;S$,T$
```

```
50  PRINT S$;T$,Pi,2*Pi
60  PRINT 2*Pi;Pi

RUN

3.14 good          bye
goodbye            3.14              6.28
6.28 3.14
^                  ^                 ^
col1               col21             col41
```

Normally , the PRINT statement causes a set of values to be printed, and then, a new line to be started. The new line start may be suppressed by ending the PRINT statement with a ";" or "," (which have the same meaning as above). Further printing by other PRINT statements will then occur where the unterminated PRINT left off. In other words,

```
10 PRINT X,Y,
20 PRINT Z,W
```

is equivalent to :

```
30 PRINT X,Y,Z,W
```

A PRINT statement with no print fields simply causes an empty line to be sent to the logical unit.

TAB function : A print field may contain TAB(arithmetic expression) instead of a literal or arithmetic expression . This causes the terminal to space until the column specified by the argument value of the TAB function is reached . If the argument value specifies a column less than the current position of the print head (or screen cursor), no spaces are produced. Note that TAB(exp) must be followed by a ";".
The example shown below will cause a sine curve to be printed onto the system line printer (LU = 2).

Example 5.3 : Drawing a sine curve.

```
10 I = 0
20 PRINT #2 TAB( 50 + 50 * SIN(I)); SIN(I)
30 I = I + .25
40 GOTO 20 \ print for ever ...
```

5.6 The DIGITS and LINE statements

General form : DIGITS = arithmetic expression
LINE = arithmetic expression

Purpose : DIGITS causes the number of digits specified by the expression to be printed in the fractionary field of a numeric value . If the expression happens to be null , then the printout format is as described under 5.5.
Values whose fractionary field includes more significant digits than the number implied by the expression will automatically be rounded (not truncated !) to the number of digits specified.
Values whose fractionary field includes less significant digits than the number implied by the expression will be extended to the right with trailing zeros. The following example presents a possible usage of the DIGIT statement.

Example 5.4 : Effect of the DIGIT statement.

```
10 INPUT A,B
20 DIGITS = B
30 PRINT A
40 GOTO 10

RUN
? 123 5
123.00000
? 1.234 2
1.23
? -9.23456E22  4
-9.2346E+22
```

The LINE statement specifies the maximum number of characters that can be printed on the same line to the logical unit. In BASIC-M, this number defaults to 80. If the print head position happens to exceed the line length specified in the LINE statement, a carriage return and line feed character will then be sent to the output device.

Comments : The DIGITS statement only affects the printout format but does not affect at all the speed of calculations.

5.7 The GOTO statement

General form : GOTO line number , or
GO TO line number

Purpose : Transfers control unconditionally to the specified statement number.

Comments : GOTO overrides the normal execution sequence of statements, and is useful for repeating a task indefinitely, or GOing TO another part of a program if certain conditions are present. GOTO should never be used for entering FOR-NEXT loops , subroutines , or user-defined procedures ; doing so may produce unpredictable results or fatal errors.

Example : The following will unconditionally pass control to statement number 20 :

```
100 GOTO 20
```

5.8 The conditional GOTO statement

General form : ON index GOTO ln1, ln2, ..., lnN

where :

- index is an arithmetic expression,
- ln1, ln2, lnN are line numbers.

Purpose : The conditional GOTO statement (also referred to as computed GOTO) transfers control to the statement whose numeric position in the list of statement numbers (reading left to right) is equal to the index possibly rounded down to a byte value. Thus , an index with a value of 3.75 would cause control to be transferred to the third statement in the list.

Comments : If the index has a value less than 1 or greater than the total number of statements listed, a runtime error is reported, and control is passed to the next statement.

Example 5.5 : Given the relationship : y=x^N, compute the value of y for 2 input values x and N , without using the exponentiation operator.

```
10  REM  this sample program does not check
20  REM  the validity of N ( 1 =< N < 5 ).
30  INPUT X, N
40  Y=X
50  ON N GOTO 90,80,70,60
60  Y=Y*X                    \ X^4
70  Y=Y*X                    \ X^3
80  Y=Y*X                    \ X^2
90  PRINT Y
95  GOTO 30
```

5.9 The GOSUB and RETURN statements

General form : GOSUB line number
RETURN

Purpose : The GOSUB and RETURN statements are used together to implement subroutines, i.e. sequences of BASIC-M statements written once in the user program that can be called for from several places.

The GOSUB statement transfers control unconditionally to the subroutine whose line number is specified.

The RETURN statement transfers control back to the the calling program, at the statement that immediately follows the associated calling GOSUB .

Comments : Subroutine calls may be nested, i.e. , a subroutine may call another subroutine, which in turn may call another one, ...etc. It is wise however to return from every subroutine called ; otherwise, the stack that stores the return addresses may overflow.

5.10 The conditional GOSUB statement

General form : ON index GOSUB ln1, ln2, ..., lnN

where index, ln1, ln2, lnN have the same meaning and effect as in the computed GOTO statement (refer to 5.8).

Purpose : The computed GOSUB statement causes control to be transferred to the statement whose numeric position in the list of line numbers is equal to the byte value of the index. As for the ON...GOTO, the index must be equal or greater than 1, and less than or equal to the total number of statement numbers in the list. The destination of the ON...GOSUB statement must be a subroutine, which as such, must end with a RETURN statement .

5.11 The IF statement

General form : IF logexp THEN statement, or
IF logexp THEN line number

where :

- logexp is a logical expression (see 2.3.5.4),

- statement is any executable statement that does not include the THEN keyword. The following are examples of invalid IF statements :

```
10 IF A>B THEN IMAGE "[C2/2]" \ Non-executable
20 IF A=3 THEN ON NMI THEN 100 \ Two THEN´s
```

Purpose : When an IF statement is executed, the logical expression is evaluated. If the relationship is true, the statement in the THEN clause is executed ; note that "THEN line number" is equivalent to "THEN GOTO line number". If the relationship is false, control is passed to the first executable statement following the IF statement.

Example 5.6 : Same definition as in example 5.5 - Modified to check that N is in a permissible range. If N is not in the range 1 thru 4, then stop execution.

```
10   INPUT X,N
15   N = INT(N) \ let N become an integer
20   IF N < 1 OR N > 4 THEN STOP
25   Y = 1
30   Y = Y * X
35   N = N - 1
40   IF N # 0 THEN 30
45   PRINT Y     \ output result
50   GOTO 10
```

Another example illustrating the call of a user-defined multi-line procedure named PLOT, based on the values of the coordinates X and Y.

```
50   IF X<313 AND X>0 AND NOT(Y>255) THEN PLOT(X,Y)
```

Comments : It must be emphasized that the IF statement causes the test of a condition (the one specified by the logical expression) at the time the IF is executed , and at this time only. On the contrary, the WHEN statement implies a continuous monitoring of a condition, as is explained in more detail further on in this manual.

5.12 The FOR and NEXT statement

General form : FOR index = exp1 TO exp2 STEP exp3
NEXT index

where :

- index is a simple arithmetic variable,
- expl, exp2, and exp3, are arithmetic expressions,
- exp3 is optional.

Purpose :

Together, a FOR statement and its paired NEXT statement delimit a FOR loop , that is, a set of BASIC-M statements that can be executed a number of times. The FOR statement marks the beginning of a loop and specifies the conditions of its execution and termination. The NEXT statement marks the end of the loop.

expl yields the initial value of the index, exp2 represents its ending value (at which the loop ends), and the amount that the index is increased or decreased after each execution of the loop is indicated by exp3.
If STEP and exp3 are omitted, an increment of 1 is assumed.

Upon execution of the FOR statement, the index is set equal to the initial value expl, then the loop is executed. When the NEXT statement is encountered, the specified increment exp3 (which may be negative) is added to the current value of the index which is then compared with the specified final value exp2. If the index is still less than (or greater than, for negative increments) or equal to the final value, the loop is executed again and the cycle continues until an increment is made that renders the index out of the specified final value. At that time, the index is set back to its final value and control falls through to the first executable statement following the NEXT statement.

Comments :

. A FOR-NEXT loop is always executed at least once.

. The final value of the index as well as the value of the increment are evaluated upon execution of the NEXT statement, and therefore, can be affected during execution of the loop.

. If the value of the increment exp3 is zero, the FOR loop executes for ever unless the value of the index is purposely set beyond the specified final value within the loop.

. Transfer out of a FOR loop is permitted, whereas transferring control into the loop may cause unpredictable results.

. FOR loops can be nested within one another as long as the internal FOR loop falls entirely within the external FOR loop ; in other words, FOR loops must not overlap ; doing so will cause an error message at compile time . The maximum number of nested FOR loops is 21.

. There must always be a NEXT statement to balance a FOR statement (with the same variable name used as index).

Example 5.7 : A third solution to the problem presented in examples 5.5 and 5.6.

```
10  INPUT X, N
15  N = INTT(N)
20  IF N <1 OR N >4 THEN STOP
25  Y = 1
30  FOR K = 1 TO N
35  Y = Y * X
40  NEXT K
45  PRINT Y
50  GOTO 10
```

Example 5.8 : Use a FOR loop to solve the problem presented in example 5.3 (draw a sine curve).

```
10  FOR I=0 TO 2*PI STEP PI/100
15  X=SIN(I)
20  PRINT #2 TAB(50+50*X);X
25  NEXT I
```

5.13 The STOP , PAUSE , and END statements

General form : STOP constant
PAUSE constant
END

where constant is a literal or numeric constant.

Purpose : The STOP and the END statements allow to terminate program execution. These two statements can appear anywhere in a program. Note that the END statement does not imply the physical end of a BASIC-M program (statements that follow an END statement are compiled).

The PAUSE statement causes program execution to be temporarly suspended ; execution resumes from the the first executable statement that follows the PAUSE statement as soon as the operator strikes any key (except a control key) on the system console.

Comments : . The END and STOP statements are not mandatory as last statement lines of a program.

. Since several PAUSE or STOP statements may exist in a program, the user may wish to be informed of where the program is running when execution is suspended or halted. This information can be provided by attaching an optional literal or numeric constant to the PAUSE or STOP statements, as shown in the next example.

Example 5.9 :

```
10 IF Temp > 280 THEN PAUSE "REPAIR COOLING"
20 WHEN Pressure > 120 THEN STOP "ALARM !!!"
30 PAUSE 22
```

5.14 Illustrative examples

This paragraph presents three sample programs that illustrate the use of some of the statements which were described in this chapter.

Example 5.10 : Compute the square root of a positive number A by using the formula :

X2 = (X1 + A/X1) / 2

The square root X2 of A is obtained by applying the above formula iteratively. X1 is initially set equal to A. The iterative process ends either when a given number of iterations have been performed , or when the absolute difference X2-X1 is less than a user-defined number.

A program is written below that inputs :

- A : the number whose square root is to be computed,
- EPS : the smallest absolute difference X2-X1 that causes the algorithm to terminate,
- ITER : the maximum number of iterations to be performed.

The program is intended to print :

- the square root of A (X2),
- the difference X2-X1,
- The amount of iterations that were performed.

Entering a value A less than or equal to 0 will stop program execution.

```
10   REM main program
20   INPUT "enter data ", A, EPS, ITER
30   PRINT                          \ empty line
40   IF A <= 0 THEN STOP            \ reject numbers <=0
```

```
50   REM call square root subroutine
60   GOSUB 120
70   PRINT "SQRT","DELTA","LOOP" \ header
80   PRINT X2, X2-X1, N          \ print results
90   GOTO 20                     \ ask for next entries
100  REM
110  REM Square root subroutine
120  X1 = A                      \ initialize X1
130  FOR N = 1 TO ITER           \ perform ITER  loops
140  X2 = .5 * ( X1 + A/X1 )     \ NEWTON's formula
150  IF ABS(X2-X1) < EPS THEN 180
160  X1 = X2                     \ X2 is new value of X1
170  NEXT N                      \ next iteration
180  RETURN                      \ exit subroutine

RUN

enter data ? 2  1.E-3  25

SQRT                DELTA               LOOP
1.41421356          -2.12341547E-06     4
enter data ?
```

Example 5.11 : Read 256 frames of a paper tape and print their binary sum.
The reader is supposed to be connected to a PIA ; the PIA control and data register are designated RDC and RDD respectively. The wiring and PIA initialization are such that :

- bit #6 of RDC is set when the tape is loaded into the reader.
- bit #3 of RDC going high causes the next frame to be read.
- bit #7 of RDC is set when a frame is available for reading.

```
10   REM PIA declaration and initialization are not shown
20   SUM = 0                    \ initialize checksum
30   REM make sure that tape is ready
40   IF RDC[6] = 0 THEN PAUSE "LOAD TAPE"
50   FOR K = 1 TO 256           \ to read 256 frames.
60   RDC[3] = 1                 \ send pulse to activate
70   RDC[3] = 0                 \ reader.
80   TIMOUT = 0                 \ initialize time-out flag.
90   IF RDC[7] = 1 THEN 130     \ increment time-out as long
100  TIMOUT = TIMOUT + 1        \ as frame is not available.
110  IF TIMOUT < 1000 THEN 90   \
120  STOP "CHECK READER"        \ time-out! stop execution.
130  SUM = SUM + RDD            \ add this frame to checksum.
140  REM reading RDD has reset RDC[7]
150  NEXT K                     \ go and read next frame.
160  PRINT "CHKSUM = ";SUM      \ 256 frames read.
```

Example 5.12 : A series of computer systems is built where each system has its own passwords . The number of passwords varies from one computer to another, and is recorded in each computer. The following sample program will allow access to a particular machine if the operator's password matches one of those defined in it.

```
10  REM line 30 defines the number and values
20  REM of the passwords for a given machine.
30  DATA 4, "H.T.IRE","FORD P.MO","N.NER","AT." \4 passwords
40  REM .....
50  FOR Try = 1 to 3
60  INPUT "Password ", Key$
70  READ N       \number of passwords accepted by this machine
80  FOR K = 1 TO N
90  READ Pass$
100 IF Key$ = Pass$ THEN 180
110 NEXT K           \ no match. read next password
120 RESTORE          \ all passwords exhausted. ask again ...
130 NEXT Try         \ unless 3 attempts already done
140 PRINT "SECURITY CHECK !"
150 PRINT CHR$(7); \ sound bell ....
160 GOTO 150         \ continuously !!
170 REM Valid password
180 PRINT "ACCESS AUTHORIZED"
```

CHAPTER 6

6. PRINT USING

6.1 General description

Paragraph 5.5 discussed the simplest form of the PRINT statement. This chapter is entirely devoted to the description of the PRINT USING statement, an extension of PRINT to perform formatted, instead of free-format outputs to the console, line printer or to a diskette file.

General form : PRINT #LU USING format , print list

where :

- LU is an expression yielding a logical unit number in the range 0 to 255 . If LU=0 or LU=1, the printout occurs on the console ; if LU=2, the printout is directed to the system line printer ; if LU>2, then the operands in the print list are saved onto a diskette file. The logical unit specification (#LU) is optional; if not present, the printout defaults to the console.

- format is a string variable name, or a string constant, or a line number of an IMAGE statement, which describes the format in which the operands in the print list are to be printed (or saved to a diskette file).

- print list is a set of string or numeric expressions (excluding the TAB function) separated by commas or semicolons ; as in the simple PRINT statement, the last item in the print list may be followed by either one of these two delimiters, which would cause the print head to stay stable after the last item in the list has been printed (no carriage return - line feed characters are sent to the printing device). In the case of the PRINT USING statement, the "," and ";" used to separate the items in the print list play no role, as long as the format string is complete enough to describe the format of all the items to print. Should the format string be incomplete, the delimiters "," and ";" will control tabbing as described in the PRINT statement (refer to paragraph 5.5).

Purpose : The PRINT USING statement is used to perform formatted outputs ; it operates by alternately outputting parts of the format string and outputting values (from left to right) from the print list. For each value in the print list , PRINT USING does the following :

The characters from the format string are printed until the format string is exhausted or until a format descriptor (enclosed in "[]"), is encountered which describes the printout format of the value to be printed.
If the format string is exhausted , free format output is used.

Before going any further into the description of the various format descriptors , let´s consider the following examples which all produce on the system line printer the formatted output shown below :

```
coll       <...  20 columns ...>       < 3> <2>

   v
   ITEM : HAMMER              COST ...  2.75
>
>        ( 2 empty lines )
```

Example 6.1 :

```
10 LET Item$ = "HAMMER"
20 LET Cost = 2.75
30 PRINT #2 USING "ITEM : [20]COST ...[3,2][/2]", Item$, Cost
```

Example 6.2 :

```
10 Format$ ="ITEM : [20]COST ...[3,2][/2]"
20 PRINT #2 USING Format$, Item$, Cost
```

Example 6.3 :

```
10 PRINT #2 USING 40, Item$, Cost
40 IMAGE "ITEM : [20]COST ...[3,2][/2]"
```

The above examples illustrate the use of three descriptors whose meaning is as follows :

[20] specifies that the string variable Item$ is to be printed in a 20-column field (left justification).

[3,2] specifies that the numeric variable Cost

is to be represented as a fixed-point number with 3 positions to print its integer part, and 2 positions for its fractionary part.

[/2] causes two carriage return - line feed strings to be sent to the printing device (the line printer in this example).

BASIC-M provides very handy format descriptors that make the language well suited for a wide variety of applications where a versatile formatting of data is at a premium. It includes many facilities of FORTRAN and COBOL, plus a few unique ones. These descriptors can also be used in conjunction with the STR$ built-in function to format memory-resident data, instead of output data.

6.2 Format descriptors

6.2.1 The Integer descriptor

General form : [k]

Purpose : Commonly used to print byte, integer, or string variables in a field of length "k".

Comments : . string variables are left-justified.

. numeric variables are right-justified; leading zeros are suppressed, the minus sign, if any, is floating (i.e. , it is "stuck" at the leftmost digit of the number).

Example 6.4 : Print 9^n (0<n<6).

```
10  FOR N=1 TO 5
20  PRINT USING 40, N, 9^N
30  NEXT N
40  IMAGE "9^[1] =[6]"

RUN

9^1 =     9
9^2 =    81
9^3 =   729
9^4 =  6561
9^5 = 59049
```

6.2.2 The string descriptor

General form : [k,option]

where : option = { R , C }

Purpose : Used to print strings only in a field of length "k".

Comments : .option "R" implies right-justification.
.option "C" implies centering within the field.

Example 6.5 : self-explanatory.

```
10 A$ = "MOTOROLA"
20 B$ = " SEMICONDUCTORS"
30 PRINT A$
40 PRINT USING "[30,R]",A$
50 PRINT A$+B$
60 PRINT USING "[30,C]",A$+B$

RUN

MOTOROLA
                      MOTOROLA
MOTOROLA SEMICONDUCTORS
   MOTOROLA SEMICONDUCTORS
                             ^
                             col30
```

6.2.3 The hexadecimal descriptor

General form : [$k]

Purpose : Used to print data which are commonly represented in the hexadecimal notation (byte or integer variable).

Comments : . the leading dollar sign "$" is not printed.
. leading zeros are printed.
. the number is right-justified.

Example 6.6 : Memory test.

```
1    INTEGER I
10   BYTE Memory(2048) ADDRESS 1024, Pattern
20   FOR Pattern = 0 TO $FF
30   FOR I = 1 TO 2048
40   Memory(I)=Pattern
50   IF Memory(I) = Pattern THEN 80
60   PRINT USING 70 , I+1023 , Pattern, Memory(I)
70   IMAGE "ADDR [$4]  WRITTEN [$2]  READ [$2]"
80   NEXT I
90   NEXT Pattern

RUN

ADDR 05FC  WRITTEN AA  READ A8
ADDR 062A  WRITTEN C2  READ C0
```

6.2.4 The horizontal spacing descriptor

General form : [Xn]

Purpose : Used to output n blanks.

6.2.5 The vertical spacing descriptor

General form : [/n]

Purpose : Used to output n strings consisting of the carriage return and line feed characters (see example 6.1).

6.2.6 The fixed-point descriptor

General form : [k,m]

Purpose : Used to output data in a format similar to the FORTRAN ordinary decimal F format.

Comments :

. "m" indicates the number of positions occupied by the fractionary part of the number (not including the decimal point).
. "k" denotes the length of the integer part, minus sign included in case of a negative value.

. the minus sign , if any , is floating.

. printout occurs in a field of length = k+m+1.

. printed numbers are rounded (not truncated).

. numbers are justified on the decimal point.

. leading zeros are suppressed.

Example 6.7 : Sine calculation.

```
10  FOR I=-Pi/2 TO Pi/2 STEP Pi/4
20  PRINT USING 40 ,I , SIN(I)
30  NEXT I
40  IMAGE "[2,4][X10][1,12]"

RUN
-1.5708           **************
-0.7854           **************
 0.0000           0.000000000000
```

```
 0.7854              0.707106781000
 1.5708              0.999999943000
```

Note that the first two sine results are not printed because one attempts to output them in the [1,12] format ; because these values are negative, "k" should be set to 2 minimum ; the printout is correct when changing the IMAGE statement to :

```
40  IMAGE "[2,4][X10][2,12]"
```

First printed line becomes :

```
-1.5708             -0.999999943000
```

6.2.7 The exponential descriptor

General form : [k,m,n]

Purpose : Used to output data in a format similar to the FORTRAN exponential E format.

Comments :

. "k" denotes the length of the integer part of the number to be printed, minus sign included if the number is negative ; the integer part always consists of one digit ; therefore the minimum value of "k" is 1 ; likewise "k" must be set equal to 2 , minimum, for printing negative numbers in this format .

. "m" is the number of positions occupied by the fractionary part (not including the decimal point).

. numbers are justified on the radix point.

. "n" denotes the length of the exponent (sign included) ; the "E" character that precedes the exponent value is not counted in "n".

. the field length of a number printed in the [k,m,n] format is k+m+n+2, where 2 reflects the 2 positions occupied by the radix point and the "E" character.

Example 6.8 : Same as example 6.7 except that the results are printed using the E format.

```
40  IMAGE "[4,6,2][X5][2,14,3]"

RUN

-1.570796E+0       -9.9999994300000E-01
-7.853982E-1       -7.0710678100000E-01
 0.000000E+0        0.0000000000000E+00
```

```
7.853982E-1       7.0710678100000E-01
1.570796E+0       9.9999994300000E-01
```

6.2.8 The commercial descriptor

General form : [Csa(sb)f1s1 ... fn(sn)fmsm ... fzsz]

Purpose : Mostly used to output data in formats similar to those of the COBOL language ; therefore, the commercial descriptor is intended for , but not limited to, business-type applications.

Comments : .the quantities "si" shown in the general form are character strings (excluding the digits 1 thru 9, parentheses, quotes and "]") reproduced "as is", except :

"!" that is printed as a blank.
"+" that is printed as a "-" if the number is negative.
"-" that is printed as a blank if the number is greater than or equal to 0.
"CR" that is printed as two blanks if the number is positive.
"DB" that is printed as two blanks if the number is positive.

.the quantities "fi" shown in the general form are integers which represent the length of a printed field for one part of a number.

.all the field descriptors "fi" are optional.

.sb , for instance DM meaning Deutsch Mark, will be printed in front of the most-significant digit.

.s1, s2 ... are printed in between the numerical fields of the integer part of the number.

.sn , which must be specified between parentheses , indicates the position of the radix point ; the radix point does not default to "." and may be represented as a string of any characters (the one specified in the parentheses) ; a null string between parentheses denotes the position of the decimal point, but does not cause any output.

.sm, ..., sz are printed in between the numerical fields of the fractionary part of the number.

.if the radix point position is not specified via the "()" indicator , it is assumed to be to the right-hand side.

.if the floating field (sb) is omitted, the integer part is printed with possible leading zeros.

.if (sb) is specified , leading zeros of the number are printed as blanks, even if sb has no characters.

.two numeric field descriptors , fi , must be separated by at least one character different from a blank.

.spaces embedded in the commercial descriptor are merely ignored.

Example 6.9 : Using the commercial descriptor. The following printout is obtained when running the program presented next :

```
As of 10/11/78
bookings are $2,190,250.75
by the end of fiscal month (10/25/78), they
should be around K$3,000 (4.800.000 SFr)

10    Book=2.19025075E+6
20    Forecast=3000
30    Date=101178
40    Current_month=10
50    Last_day=25
60    Year=78
70    REM
80    PRINT USING 90, Date, Book
90    IMAGE "As of [C2/2/2][/]bookings are [C($)1,3,3(.)2]"
100   PRINT "by the end of fiscal month ";
110   PRINT USING 120, Current_month, Last_day, Year;
120   IMAGE "([2]/[2]/[2]), they[/]should be "
130   PRINT USING 140, Forecast, 1600*Forecast
140   IMAGE "around [C(K$)1,3] ([C1.3.3 ! SFr])"
```

Example 6.10 : Another sample program using the commercial format.

```
             :
             :
100   PRINT "STATEMENT OF ACCOUNT - MONTH : "; Month$
110   FOR I=1 TO Nb_transac
120   INPUT #7, Date, Sum
130   PRINT USING 200, Date, Sum
140   NEXT I
             :
200 IMAGE "[X6][6][X15][C($)3,3,3(!!!)2 !!!DB]"
```

Data read from diskette file :

```
050779  ..... -500
051579  ..... -215.75
051779  ..... 9000
052279  ..... -410
052579  ..... 3500.50
```

Printout obtained :

```
 col7                    col36
    v                        v
     50779                $500    00    DB
     51579                $215    75    DB
     51779              $9,000    00
     52279                $410    00    DB
     52579              $3,500    50
```

CHAPTER 7

7. DECLARATION STATEMENTS

This chapter describes the several statements used to define the type, structure, and address of the user's program data, as well as those used to declare assembly-language subroutines.

7.1 Declaring BYTE variables

Byte variables are declared via the BYTE statement whose general form is as follows :

BYTE V1(x1,y1) ADDRESS a1, ... , Vn(xn,yn) ADDRESS an

where :

.Vi represent the names of byte variables.
.xi, yi are unsigned decimal constants denoting the size of the dimensions of the byte array variable Vi.

.ai indicates the memory address of the variable Vi.

Comments :

.xi, yi are optional ; if present, they specify the size of the first and second dimensions respectively of the byte array Vi. For one-dimensional byte arrays (byte vectors), xi must not exceed 65535; for two-dimensional byte arrays, xi and yi must each be less than or equal to 255.

.the address clause " ADDRESS ai " is optional; if present, it defines the memory address ai of variable Vi. ai is an unsigned decimal or hexadecimal constant, or the name of an already declared variable to which Vi is to be equated.

.the keyword "ADDRESS" may be abbreviated as "ADDR".

.the BYTE keyword applies to all the variables of the statement line.

Example 7.1 : Declaring byte variables.

```
10  BYTE PIA(2) ADDR $8008, Memory(255,16)
20  BYTE PIA_data ADDRESS PIA
30  BYTE Var1, Var2 ADDR Var1, Var3
```

.PIA represents a 2-byte vector based at the absolute address $8008.
.Memory defines a matrix of 255 rows by 16 bytes.
.PIA_data represents the first item of the 2-byte vector PIA, so PIA_data is located at the absolute address $8008.
.Var1, Var2, Var3 are all three simple byte variables ; Var2 is equated to Var1 , therefore Var1 and Var2 reside at the same address. Note that Var2 is equated to an already declared variable (backward reference).

7.2 Declaring INTEGER variables

Integer variables are declared by using the INTEGER statement whose general form is :

INTEGER V1(x1,y1) ADDRESS a1, ... , Vn(xn,yn) ADDRESS an

where Vi, xi, yi, and ai have the same meaning as for the BYTE statement.

Comments : .same as under 7.1.

.the INTEGER keyword applies to all the variables defined in the statement line.

Example 7.2 : Declaring integer variables.

```
10  BYTE Pia_D ADDR $8008, Pia_C ADDR $8009
20  INTEGER PIA ADDRESS Pia_D, Word(16,16)
30  REM Pia initialization - solution #1
40  Pia_D = $FF
50  Pia_C = $4          \ or Pia_C[2] = 1
60  REM alternate solution
70  PIA = $FF04
```

7.3 The DIM statement

The DIM statement is used for declaring vectors or arrays of real or character data; its general form is as shown below :

DIM V1(x1,y1) ADDRESS a1 , ... , Vn(xn,yn) ADDRESS an

where :

.Vi are variable names; a variable name ending with a dollar sign ($) defines the variable it represents as a character variable; any other name specified in a DIM statement defines a real variable.
.xi, yi are unsigned decimal constants denoting the first, respectively second size of the dimensions of the variable Vi.

.ai is the memory address of the variable Vi.

Comments : .xi, yi are optional. For one-dimensional arrays (vectors), xi must not exceed 65535. For two-dimensional arrays, xi and yi must not exceed 255.

.the address clause "ADDRESS ai" is optional ; if present , it defines the memory address of the variable Vi. ai is an unsigned decimal or hexadecimal constant less than 65536, or the address of an already declared variable to which Vi is to be equated.

.the keyword "ADDRESS" may be abbreviated as "ADDR".

.simple character or real variables which are to be equated to an absolute address or to the address of another pre-declared variable must be declared with a DIM statement; in addition, these variables must be explicitly declared as having one dimension of size equal to one. Thus, if a simple real variable V is to be defined at the address, say, 1024 , one should declare it with the statement line :
10 DIM V(1) ADDRESS 1024

Example 7.3 : Declaring variables via the DIM statement.

```
10  BYTE Display(22,80) ADDRESS $E000
20  DIM Screen(22,16) ADDR Display, Varl, Var(2,3)
30  DIM A$(1) ADDRESS Display, TEXT$(4,5)
```

.In this example, the real matrix "Screen" is equivalenced with the byte matrix "Display" : therefore , there is no difference, as far as storage address is concerned, between these two matrices; however, referencing an item of matrix "Screen" will also reference 5 items of matrix "Display" (since a real variable occupies 5 bytes).
.Line 30 says that the character variable A$ is to occupy the first 32 bytes of matrix "Display".

7.4 Declaring external subroutines

The absolute address of a user-written assembly language procedure or function must be declared explicitly prior to being called for. The general form of such a declaration is as follows :

EXTERNAL P1 ADDRESS a1 , ... , Pn ADDRESS an

where :

.Pi represents the name of the user-supplied assembly language procedures / functions.

.ai is an unsigned decimal or hexadecimal constant denoting the absolute starting address of the procedure / function Pi.

Comments : .the keyword "EXTERNAL" can be abbreviated as "EXT".

.the keyword "ADDRESS" can be abbreviated as "ADDR".

.the "EXTERNAL" or "EXT" keywords apply to all the subroutines declared in the statement line.

Example 7.4 : Declaring external subroutines.

```
10  EXTERNAL XPCRLF ADDRESS $F021, XORBUG ADDR $F02D
20  EXT XPSPAC ADDRESS $F02A
```

7.5 Runtime initialization

When the "RUN" command is invoked, all the variables defined in a BASIC-M source program are initialized to zero; this rule, however, does not apply to variables which are equated to absolute memory addresses (variables declared with an "ADDRESS" or "ADDR" specification). For example, given the statement line :

```
BYTE A(255) ADDR 1024, B , C(10) ADDR $FF , D$(20)
```

The simple variable B, as well as the string vector D$, will be all cleared upon execution, whereas the vectors A and C will be left unchanged.

CHAPTER 8

8. REAL-TIME MONITORING

As stated earlier, BASIC-M was specified so as to be a high-level language which had yet to provide facilities for the user to work close to the target environment . Not surprisingly, the language includes the necessary statements to monitor hardware events such as interrupt requests to the MC6809 processor and keystrokes. This chapter discusses those along with the statements to allow for a software monitoring of runtime conditions and errors.

8.1 The ON interrupt THEN statements

There are three statements which allow the user to enable and process the three possible interrupt requests to the MC6809 processor ; their syntax is as follows :

General form :
```
ON NMI THEN action
ON IRQ THEN action
ON FIRQ THEN action
```

where :

.NMI refers to the non-maskable interrupt request to the MC6809 processor ; the processor NMI input is edge-sensitive.

.IRQ and FIRQ refer to the interrupt request, respectively fast interrupt request to the MC6809 ; the corresponding inputs are both level-sensitive ; therefore , the associated interrupt handlers must cancel the original interrupt source so that the interrupted program can resume execution once the interrupts have been serviced.

Comments :

.interrupts must be enabled via one of the above statements in order to be recognized and processed ; should an interrupt request occur in the system , which has not been previously enabled by its corresponding "ON" statement , the runtime package will flag it as a spurious interrupt and will abort the execution of the BASIC-M program.

.the "THEN" clause indicates the service to be provided when the corresponding interrupt request is detected. "action" must be an executable statement with the following exceptions :

- .the "FOR" statement.
- .the "GOTO" statement.
- .any executable statement that includes a THEN clause (IF, WHEN, ON).

.the action routines which include more than one BASIC-M line of code must be structured like subroutines , that is, they must end with a "RETURN" statement. Therefore , subroutine and procedure calls are permitted in an "ON interrupt" statement.

.interrupt requests are disregarded further to the execution of an associated "NEVER interrupt" statement. Should they still occur, they are treated as spurious interrupts (runtime fatal error).

Example 8.1 : Speed calculation.
A disk is mounted on the shaft of an engine, which has an index hole delivering a pulse on a control line of a PIA at every revolution. This latter drives the processor IRQ input. The program listed below records the maximum speed of the engine (r.p.m). A MC6840 timer is programmed to request an NMI interrupt every 20 ms. The PIA and timer initialization routines are not shown.

```
10   GOTO 110        \ skip over procedure definition
12   REM real-time clock interrupt
14   REM service routine .........
16   REM -------------------------
18   DEF Check_Time
20   Time=Time+1
22   IF Time < 50 THEN RETURN
24   Time=0
26   RPM=Speed*60
28   IF RPM > Max THEN Max=RPM
30   Speed=0
32   RETURN
34   REM
100  REM **** MAIN PROGRAM ****
110  Time=0         \ Time count
120  Max=0          \ maximum speed
130  Speed=0        \ current speed (rpm)
140  ON NMI THEN Check_Time    \ enable NMI
150  ON IRQ THEN GOSUB 240     \ enable IRQ
160  Init_Timer     \ procedure to set up timer
170  Pia[0]=1       \ enable sampling
180  GOTO 180       \ program does nothing but waits
190  REM            \ for interrupts.
200  REM **** END OF MAIN ****
210  REM Interrupt routine to be serviced
220  REM on occurence of the index hole..
230  REM --------------------------------
240  Dummy = Pia \ reset interrupt source
250  Speed = Speed + 1
260  RETURN
```

8.2 The ON KEY statement

User-defined keys allow the operator to interrupt an active program to run a higher priority subprogram with a single keystroke. Function key interrupts are enabled on execution of the "ON KEY" statement whose general form is as follows :

General form : ON KEY k1, k2, ..., kn THEN action

where :

.ki are arithmetic expressions rounded to byte values which indicate which function keys are to be considered as active. Striking any key of the list k1,k2,...,kn will cause the execution of the statement specified in the THEN clause (action). This statement must obey the same rules as the ones set in the previous paragraph. Keystrokes of function keys not previously enabled by an "ON KEY" statement are merely ignored.

Comments :

.each arithmetic expression ki must result in a value greater than 0 and less than 17 (16 keys). A runtime error message is reported if this is not the case.

.striking an active key causes an NMI request to the MC6809 processor ; this , however, does not mean that the "ON KEY" statement hinders usage of the other interrupt-related statements described under 8.1 : they all can co-reside in a BASIC-M program.

.since there may be several keys enabled by a single "ON KEY" statement , it might be desirable to know which key was depressed last ; this information is supplied by the built-in function "FKEY" that returns a value between 1 and 16 which denotes the number of the last depressed key.
.the value returned by "FKEY" is only meaningful after execution of the "ON KEY" statement. The value of FKEY is zero if no function key has been activated since the last call to FKEY ; in other words, reading FKEY causes it to be resetted to zero.

.function keys enabled by the "ON KEY" statement can be further individually disabled (desactivated) by using the associated "NEVER KEY" statement (see paragraph 8.5).

Example 8.2 : Same as example 8.1, but modified to print the variable "Max" whenever function key 16 is activated.

```
180 ON KEY 16 THEN PRINT USING "Speed = [I3] rpm", Max
185 GOTO 185
```

Example 8.3 : 5 different tasks are initiated by striking the function keys F1 thru F5. Write a program to dispatch control on a valid keystroke.

```
10  ON KEY 1,2,3,4,5 THEN ON FKEY GOSUB 100,200,300,400,500
     :
100 REM - task #1 -
     :
190 RETURN
     :
500 REM - task #5 -
     :
560 RETURN
```

The above program is equivalent to :

```
10  ON KEY 1 THEN GOSUB 100
20  ON KEY 2 THEN GOSUB 200
30  ON KEY 3 THEN GOSUB 300
40  ON KEY 4 THEN GOSUB 400
50  ON KEY 5 THEN GOSUB 500
```

8.3 The WHEN ... THEN statement

The statements described sofar in this chapter are all used to perform a hardware monitoring of external events , since they are based on the processor interrupt capabilities. The next two statements are aimed at easing the continuous testing of software conditions during program execution.

The WHEN statement is used to regain program control when a user-defined condition is satisfied ; its general form is :

General form : WHEN logexp THEN action

where :

.logexp stands for a logical expression which specifies the condition to be continuously tested. Logical expressions are discussed in paragraphs 2.3.5.4 and 4.6.

.action is an executable statement which conforms to the rules set under 8.1.

Comments : .any number of WHEN statements can appear in a program, but only the last executed is effective at any time.

.unlike an "IF" condition which is only tested upon execution of the "IF" statement, a "WHEN" condition is tested prior to executing each and every line of the program. The condition monitoring is initiated on occurence of a WHEN statement ; should the condition become satisfied at any time during program execution , control is then transferred to the action routine ; during execution of this latter routine , the condition monitoring is temporarily suspended up until the action has been wholly executed ; the action terminates when its RETURN statement is encountered (in case of a multi-line action routine). This functionning is illustrated on the following example :

```
10  WHEN A>100 THEN GOSUB 50
20  INPUT A
30  A=A+25
40  GOTO 20
50  PRINT "ACTION *** SQR(A) = ";
60  PRINT SQR(A)
70  RETURN

RUN
? 144
ACTION *** SQR(A) = 12  (due to line 30)
ACTION *** SQR(A) = 13  (due to line 40)
ACTION *** SQR(A) = 13  (due to line 20)
? 69
?
```

Note that the condition is not tested during the action routine which consists of the lines 50, 60 , and 70. The action would otherwise be re-entered for ever.

.care should be exercised so as not to run for ever in the action routine ; this would happen if a WHEN condition , once met , is never rendered false further in the program. In the above example , the condition is related to the input value A which is a subject to change ; thus , depending on the input value, the condition will be sometimes satisfied, sometimes unsatisfied. If, in the same program, line 20 would be changed to "A=200" , it is obvious that the action associated with the WHEN statement would be re-entered continuously. The sample program which follows is another example of a situation where the condition, once satisfied, will no longer be rendered false ; consequently, the program will not work as expected. The program is supposed to count the number of keystrokes (function keys excluded). The keyboard strobe signal is connected on a control line of a PIA labelled KEYBC. A keystroke sets the most-significant-bit of the byte KEYBC to one.

```
10  BYTE KEYBC ADDRESS $EF83
20  Count=0    \ initialize count
30  WHEN KEYBC[7]=1 THEN Count=Count+1
40          :
```

Whatever the number of keystrokes (but at least one), the result Count will be erroneous for bit 7 of KEYBC , once set, will never be reset. This is due to the operating principle of the PIA, which requires that its data register be read to reset the most-significant-bit of its control register. The reader is encouraged to go through the example 8.5 in order to avoid this common pitfall.

.there are circumstances however, where a WHEN condition disappears by itself ; this is the case when variables defining the condition are related to hardware random signals. The following example presents such a situation ; the program is intended to drive an audible alarm for as long as a door is open.

```
10  WHEN Door[3]=1 THEN Bell
```

The bit indicating the state of the door depends on an external condition which, as such, cannot be controlled by program.

.The execution of the "NEVER WHEN" statement disables all WHEN requests .

.A WHEN statement can be embedded in the action routine of another WHEN statement, according to the following scheme :

```
WHEN condition1 THEN action1
     .
     .
action1 :       .
                .
          WHEN condition2 THEN action2
                .
return1 :  RETURN
```

If this structure is implemented, condition2 will be monitored after, and after only, that condition1 has been fulfilled AND that action #1 subroutine has returned. This scheme provides for switching of conditions. The sample program presented next illustrates the functionning just described :

```
10  WHEN A>100 THEN GOSUB 200
20  INPUT A
30  GOTO 20
      :
200 PRINT "LINE 10 WHEN ACTIVE"
210 WHEN B>100 THEN PRINT "LINE 210 WHEN ACTIVE"
220 B=A
230 B=B+1
240 RETURN

RUN
? 110
LINE 10 WHEN ACTIVE    ( due to line 30 )
LINE 210 WHEN ACTIVE   ( due to line 20 )
? 2
LINE 210 WHEN ACTIVE   ( B remains unchanged and equal
LINE 210 WHEN ACTIVE     to 111 since conditionl is no
?                        longer tested. )
```

.usage of the WHEN statement results in some degradation as far as program execution speed is concerned.

.programs that make use of the WHEN statement should not be compiled with the "S" compiler option ; this option prevents the compiler from generating the statements which allow runtime condition testing. (refer to "sytem commands").

Example 8.4 : Effect of the WHEN statement.

```
10  WHEN X>144 AND X<200 THEN PRINT SQR(X)
20  INPUT X
30  GOTO 20

RUN
? 30      ( WHEN condition not satisfied )
? 169     ( WHEN condition satisfied )
13        ( due to execution of line 30 )
13        ( due to execution of line 20 )
?
```

Example 8.5 : Video game. The program listed below moves a ball along the first top line of the EXORset display (alphanumeric memory based at address $E000). The game consists in intercepting the ball when it is in the middle part of the line (column 39 thru 41). A shot is made by striking any key on the system console. The keyboard is connected to a PIA based at the address $EF82 ; striking a key sets the most significant bit of the control register at address $EF83.

```
10  BYTE KEYBD ADDR $EF82, KEYBC ADDR $EF83
20  BYTE LINE(80) ADDRESS $E000
30  GOTO 110
40  REM --- procedure to check the position of
50  REM --- the ball on occurence of a shot ..
60  DEF Check_shot
70  Dummy=KEYBD                     \ reset WHEN condition
80  IF J>38 AND J<42 THEN STOP "You won"
90  RETURN
100 REM --- main program
110 MAT LINE=SET[ASC(" ")]           \ erase display line
120 WHEN KEYBC[7]=1 THEN Check_shot      \ set condition
130 K1=1                            \ set line boundaries
140 K2=80
150 DK=1                            \  and step for move
160 FOR J=K1 TO K2 STEP DK          \move ball along line
170 LINE(J)=ASC("O")
180 FOR BB=1 TO 10                  \      software delay
190 NEXT BB
200 LINE(J)=ASC(" ")
210 NEXT J
220 TEMP=K1                        \ exchange boundaries
230 K1=K2                          \ to move the ball in
240 K2=TEMP                        \ the reverse direction
250 DK=-DK
260 GOTO 160
```

Example 8.6 : Using the WHEN statement to replace several IF statements.
When several identical tests have to be done in a BASIC program, a WHEN statement can do the job economically, as illustrated below :

```
Classical method                   Using WHEN
----------------                   ----------
10  INPUT A                        10  WHEN A>100 THEN STOP
20  IF A>100 THEN STOP             20  INPUT A
      :                                  :
70  A=A+10                         70  A=A+10
80  IF A>100 THEN STOP                   :
      :                                  :
110 A=A*B                          110 A=A*B
120 IF A>100 THEN STOP                   :
      :                                  :
```

8.4 The ON ERROR statement

Normally, a runtime error causes the associated error message to be displayed, and program execution to be aborted in case of a fatal error. The ON ERROR statement provides a means to process non-fatal errors only. User-defined error processing may vary from a simple translation of the error message in the user´s native language, to a more sophisticated error recovery action. The syntax of the ON ERROR statement is as follows :

General form : ON ERROR THEN action where :

.action is an executable statement that conforms to the rules set in paragraph 8.1.

Comments : .the runtime error codes defined in BASIC-M are listed in Appendix ?. The ERR function returns the code of the last error that occured . Calling the ERR function automatically resets its value to zero.

.user handling of errors is activated on execution of the "ON ERROR" statement, while the "NEVER ERROR" statement disables it and therefore causes the normal error processing to resume.

.the statement :

```
10   ON ERROR THEN RETURN
```

causes all subsequent errors to be merely ignored (error messages are not displayed).

Example 8.7 : Translation of error messages.

```
10   DIM ER$(33),A(5,5)
20   DATA "DIVISION ENTIERE PAR 0"
30             :
80   DATA "TRANSPOSITION ILLEGALE"
               :
100  FOR I=1 TO 33
110  READ ER$(I)
               :
230  ON ERROR THEN GOSUB 500
               :
340  MAT A=INV(A)
               :
               :
500  PRINT ER$(ERR)
510  RETURN

RUN
INVERSION DE MATRICE SINGULIERE
```

Example 8.8 : Error recovery.
The program below fills a buffer with data, then transfers it to a diskette file. The ON ERROR statement is used to detect when the buffer is full (attempting to store a data

beyond the buffer end causes an error).

```
10  ON ERROR THEN GOSUB 200
20  I=0
30  INPUT A
40  IF A=0 THEN 100
50  I=I+1
60  Buffer(I)=A
70  GOTO 30
     :
100  :
     :
200 MAT PRINT #5 Buffer
210 I=1
220 Buffer(1)=A
230 RETURN
```

8.5 The NEVER statement

For each of the ON statements discussed in this chapter , there is a paired NEVER statement whose function is to cancel a monitoring request. The several NEVER statements are as follows :

General form : NEVER NMI
NEVER IRQ
NEVER FIRQ
NEVER KEY k1,k2,...,kn
NEVER WHEN
NEVER ERROR

where :

.k1,k2,...,kn have the same meaning as described under 8.2.

Example 8.9 : Disabling selected function keys.

```
10  K=1                \ default execution to task 1
20  ON KEY 1,2,3,4,5 THEN GOSUB 100
      :
      :
50  TASK(K)                        \ run kth task
      :
100 K=FKEY  \ read number of function key just hit
110 FOR I=1 TO K     \ disable keys of lower order
120 NEVER KEY I
130 NEXT I
140 RETURN
```

8.6 More about the RETURN statement

As stated earlier, it is mandatory to end a multi-line action subroutine associated with an ON statement, with a RETURN statement. In the particular context of real-time monitoring , where an action can be considered as a service routine which is activated randomly (based on hardware or software conditions being met), RETURN plays the same role as the instruction RTI which is used to return from an interrupt service routine. Therefore, its effect is not the same as the RETURN statement that terminates a BASIC-M subroutine. To highlight this distinction, let´s consider the following program :

```
10  WHEN I=4 THEN RETURN
        :
50  I=0
60  GOSUB 100
        :
100 PRINT I
110 I=I+1
120 GOTO 100
```

The above program will run for ever in the loop delimited by the lines 100 and 120 ; when I reaches the value 4, control is transferred to the action routine which does not absolutely nothing (the RETURN statement does not imply any concrete action). The point is that the RETURN keyword appearing in line 10 is not at all connected with the execution of the subroutine at line 100.

Another use of the RETURN statement was pointed out in paragraph 8.4. Given the following :

ON ERROR THEN RETURN

All non-fatal errors are ignored (no error message is displayed).

Finally, the statement :

ON KEY kl,...,kn THEN RETURN

allows to activate the FKEY function so that it further returns the number of the last key hit ; not using the ON KEY statement would result in FKEY being always equal to zero. The user however, must remember that calling the FKEY function automatically resets its value to zero.

CHAPTER 9

9. MATRIX OPERATIONS

This chapter explains matrix manipulation. It is intended to show the matrix capabilities of BASIC-M and assumes that the programmer has some knowledge of matrix theory.

Matrix definition, declaration, type, and arrangement in storage are topics which have already been discussed in chapters 3 (paragraph 3.5) and 7. This section only describes the types of operations which can be performed on matrices.

9.1 The classical approach

The classical approach to solving problems in which matrix operations are involved, consists in operating on every item of the matrix. This is because most BASIC´s, especially those implemented on microcomputers , do not make provision for considering a matrix as an entity ; as a result , it is the programmer´s responsibility to take care of array indexing and to find out, if needed, the statements to translate complex mathematical algorithms such as those involved in matrix inversion. To cope with the tasks just mentioned, the programmer needs to write several statement lines. For instance, if all the elements of a two-dimensional array A(4,7) are to be set to a given value, say zero, the most efficient program is likely to look like :

```
10  FOR I=1 TO 4
20  FOR J=1 TO 7
30  A(I,J)=0
40  NEXT J
50  NEXT I
```

In BASIC-M , the same problem can be solved by using the single statement :

```
10  MAT A = ZER
```

Clearly , the classical solution translates into a much larger machine code (10 times larger than the one yielded by solution #2 !!!). Not suprisingly the first solution will execute much slower than the second; this is mainly due to the fact that array indexing is handled at a "high level" (BASIC statements) as opposed to the index calculations yielded by the second program, which are handled by the runtime package assembly-language instructions.
The subsequent paragraphs detail the operations which can be carried out on matrices using BASIC-M matrix-oriented

statements.

9.2 The MAT READ statement

General form : MAT READ Arr

where :

.Arr is the name of a one- or two-dimensional numeric or character array.

Purpose : To fill the entire matrix from the current DATA statement in the row, column order: 1,1; 1,2; 1,3; etc.

Comments : .The number of elements read is controlled by the implicit or explicit statements that specify the matrix size.

.The MAT READ statement conforms to the same rules as the simple READ statement (see paragraph 5.3).

Example 9.1 : Using MAT READ.

```
10  DIM Name$(2,2),A(2,3)
20  BYTE Value(4)
30  DATA "JOHN","MARY","KATE","LEE",$FF
40  DATA 2, $41, 5, 2.718
50  MAT READ Name$
60  MAT READ Value
70  READ Constant
     :
```

The following assignment is to take place :

```
     Name$(2,2)          Value(4)         Constant
     ----------          --------         --------
                     !                  !
JOHN       MARY      !  $FF  2  $41  5  !  2.718
KATE       LEE       !                  !
```

9.3 The console MAT INPUT statement

General form : MAT INPUT prompt , Arr
or
MAT INPUT Arr

where :

.prompt is an optional literal constant.
.Arr is the name of a one- or two-dimensional numeric or character array.

Purpose : To assign values to array elements from the keyboard without specifying each array element individually.

Comments : .A more complex form of the MAT INPUT statement is described in chapter 12 , which allows data to be input from a diskette file.

.When a MAT INPUT statement is executed, it causes a question mark to be displayed, possibly preceeded by the input prompt, if any. The operator is then requested to enter a list of values that will be assigned row-by-row to the elements of the specified array. Data to be assigned to one-dimensional arrays (vectors) can be input on the same line.

.Control codes, string input, and error checking are the same as described under 5.4 (console INPUT statement).

Example 9.2 : Using the MAT INPUT statement.

```
10  INTEGER A(3,2)
20  DIM M$(3)
30  MAT INPUT "enter coefficients ", A
40  PRINT "enter array M$"
50  MAT INPUT M$

                                notes
RUN                             -----
enter coefficients ? 1 2 3       (1)
? 4 (CTRL-X)                     (2)
? 6 H                            (3)
    ^
RETYPE FROM ARROW
? 7                              (4)
? !                              (5)
enter array M$
? "How " "are " "you"            (6)
```

---

(1) - The third value ,3 , is ignored as data are entered row-by-row and A is defined as having two items per row.
(2) - Typing CTRL-X cancels the input data of the entire row (the second row in this example).
(3) - Still inputting second row of matrix A. H is not a numeric data, hence the error message. Note that the value 6 was assigned to A(2,1).
(4) - 7 completes the input of data assigned to the second row. 7 goes into A(2,2).
(5) - Because a "!" was entered, the items of the third row remain unchanged (equal to zero in our case).
(6) - The data assigned to the string vector M$ are entered on the same line ; note that

the delimiting quotes can be freely separated from each other by a blank or a comma.

The matrix A and the vector M$ now contain the following data :

A(3,2)		M$(3)
1	2	How
6	7	are
0	0	you

9.4 The MAT PRINT statement

General form : MAT PRINT #LU Arr

where :

.LU is an arithmetic expression which specifies the output port.
.Arr is the name of a one-dimensional or two-dimensional numeric or character array.

Purpose : To display the elements of a specified array without referring to each array element individually.

Comments : .The "#LU" clause is optional. When not specified, printout is directed to the console. So is it when LU is rounded to a byte value equal to 1. When LU is equal to 2, printout occurs on the system line printer. If the evaluation of LU leads to a value between 3 and 255, the array elements are directed to a diskette file. Using a diskette file to store matrices is discussed in chapter 12.

.When using the MAT PRINT statement, the format of all displayed values is standardized, so is the spacing between values on the same line.

- Numeric array elements are printed in free-format. An array is displayed row-by-row; the following rule applies: the elements of a row are displayed on the same line as long as the line length defined by the "LINE" statement is sufficient to accommodate them. If the length is not sufficient, a new line is started to print the remaining elements of the row. Then an empty line is displayed prior to printing the elements of the next row. Each array element uses one full print zone of 20 characters.

- String array elements are displayed row-by-row. The elements of a row are concatenated prior to being printed. Should the line length defined by the "LINE" statement be insufficient to accommodate the resulting string to print, a new line is started, which is not further followed by an empty line (as opposed to the way numeric arrays are displayed).

.Upon execution of the MAT PRINT statement, the carriage return and line feed characters are sent to the output device.

Example 9.3 : Using MAT PRINT.

```
10  DIM A(3),B(2,5),C$(3,6)
20  DATA 1,2,3
30  DATA 4,5,6,7,8
40  DATA 9,10,11,12,13
50  DATA "A","B","C","D","E","F"
60  DATA "G","H","I","J","K","L"
70  DATA "C$(3,1)","C$(3,2)",".",".",".","."
80  MAT READ A
90  MAT READ B
100 MAT READ C$
110 MAT PRINT A
120 MAT PRINT B
130 MAT PRINT C$

RUN                 col21               col41
                    v                   v
1                   2                   3


4                   5                   6                   7
8
9                   10                  11                  12
13


ABCDEF
GHIJKL
C$(3,1)C$(3,2)....
```

9.5 Copying a matrix

General form : MAT Arr1 = Arr2

where :

.Arr1 and Arr2 are names of one- or two-dimensional numeric or character arrays.

Purpose : To assign the elements of one array (Arr2) to another array (Arr1).

Comments : .Arr1 and Arr2 must be of the same type, i.e., either both numeric, or both character.

.Arr1 and Arr2 must have identical dimensions, but the size of their dimensions need not to be the same. However, the number of elements in Arr2 must not exceed the number of elements in Arr1. The following are examples of matrix assignments :

```
10  DIM  A(3,2), B(9)
15  BYTE C(2,4),D(8)
20  DIM M1$(3), M2$(2,2)
25  REM .......... valid assignments .......
30  MAT C = A
35  MAT B = D
45  REM ......... invalid assignments ......
50  MAT C = A      \ dimensions not identical
55  MAT C = M1$    \ mixed-type assignment
```

.After execution of the assignment, Arr1 assumes the same dimensions as Arr2. For instance, executing the statement at line 30 changes the dimensions of array C from (2,4) to (3,2). Likewise, statement 35 changes the dimension of B from (9) to the dimension of D, i.e., (8).

.Numeric conversions are performed when Arr1 and Arr2 are not of the same numeric data type (see next example).

Example 9.4 : Matrix copy.

```
10  DIM  A(3,2)
15  BYTE B(2,4)
20  MAT INPUT A
25  PRINT
30  MAT INPUT B
35  PRINT "MATRIX B BEFORE ASSIGNMENT"
4?0  MAT PRINT B
45  MAT B = A
50  PRINT "MATRIX B AFTER ASSIGNMENT"
55  MAT PRINT B
```

```
RUN
? 1 2                ( entering A )
? 255 3
? 10 20

? 254 3 2 1          ( entering B )
? $AA 9 10 20
MATRIX B BEFORE ASSIGNMENT
```

FE	3	2	1
AA	9	A	14

MATRIX B AFTER ASSIGNMENT

1	2
FF	3
A	14

9.6 Matrix addition and subtraction

General form : MAT Arr1 = Arr2 oper Arr3

where :

.oper is the plus sign "+" to perform matrix addition, or the minus sign "-" to perform matrix subtraction.
.Arr1, Arr2, Arr3 are names of one- or two-dimensional numeric arrays.

Purpose : To add or subtract the contents of two matrices Arr2 and Arr3 and assign the result to a third matrix Arr1.

Comments : .The three arrays involved in this statement must have identical dimensions (after re-dimensioning, if any). In addition, the size of the dimension(s) of Arr2 and Arr3 must be identical.

.Arr1 is re-dimensioned to the size(s) of Arr2 (or Arr3 as they must be the same).

.All three matrices must be numeric.

Example 9.5 : Matrix addition.

```
10  DIM A(3,3), B(2,2)
20  INTEGER C(2,2)
30  MAT INPUT B
40  PRINT
50  MAT INPUT C
60  MAT A=B+C     \ A is re-dimensioned to (2,2)
70  MAT PRINT A

RUN
? 1 2  ( entering B )
? 3 4

? $FFFF 10                 ( entering C )
? 20 30

0                   12      ( result A )
23                  34
```

9.7 Matrix multiplication

General form : MAT Arr1 = Arr2 * Arr3

where :

.Arr1, Arr2, and Arr3 are names of one- or two-dimensional numeric arrays.

Purpose : To perform the mathematical matrix multiplication of two numeric matrices Arr2 and Arr3 and assign the product to a third matrix Arr1. In matrix multiplication, a matrix A of dimensions (p,m) and a matrix B of dimensions (m,n) yield a product matrix C of dimensions (p,n) such that for i=1,2,...,p, and for j=1,2,...,n:

C(i,j) = sum of [A(i,k)*B(k,j)] for k=1,2,...,m

Below is an illustration of the application of this formula to square matrices (2,2).

A		B		C = A * B	
a	b	e	f	a*e+b*g	a*f+b*h
c	d	g	h	c*e+d*g	c*f+d*h

Comments : All of the following relationships must be true (after re-dimensioning, if any) :

-The number of columns in the second matrix Arr2 must equal the number of rows in the third matrix Arr3.

-The number of rows in the first matrix (product matrix Arr1) must equal the number of rows in the second matrix Arr2.

-The number of columns in the product matrix Arr1 must equal the number of columns in the third matrix Arr3.

-Matrix Arr1 may be one-dimensional if either p or n is equal to 1.

-When Arr2 is a one-dimensional array, it is treated as an array consisting of one row.

-When Arr3 is a one-dimensional array, it is treated as an array consisting of one column.

-The same array variable name must not appear on both sides of the "=" sign; this would yield an erroneous result although no error is reported.

-Mathematically, Arr2*Arr3 is not equal to Arr3*Arr2.

Example 9.6 : Matrix multiplication.

```
10  DIM A(2,3),B(3,4),C(2,4)
15  MAT INPUT A
20  PRINT
25  MAT INPUT B
30  MAT C=A*B
35  MAT PRINT C

RUN
? 1 2 3              ( entering A )
? 4 5 6

? 10 11 12 13        ( entering B )
? 14 15 16 17
? 18 19 20 21

92                   98                  104                 110
218                  233                 248                 263
```

9.8 Scalar operations

General form : MAT Arr1 = Arr2 oper (exp)

where :

.Arr1 and Arr2 are names of one- or two-dimensional numeric arrays.
.oper is one of the following operators : +,-,*,/.
.exp is an arithmetic expression which must be specified between parentheses.

Purpose : Scalar addition and subtraction allow an arithmetic expression to be added to, or subtracted from all the elements of the array Arr2, and to store the results in the array Arr1.
With scalar multiplication or division, each element of the array Arr2 is multiplied or divided by the specified arithmetic expression, and the results are stored in the array Arr1.

Comments : .Arr1 and Arr2 must be either both one-dimensional, or both two-dimensional arrays.

.The size of the dimensions of Arr1 and Arr2 need not be identical; however, the number of elements in Arr2 must not exceed the number of elements in Arr1 (after re-dimensioning, if any).

."MAT Arr1 = Arr1 oper (exp)" is a valid operation.

.After execution, Arr1 is re-dimensioned to the size of the dimensions of Arr2.

.Numeric type conversions are performed, if required (see example 9.7).

Example 9.7 : Scalar division.

```
10  BYTE A(3,3)
20  DIM B(2,4)
30  MAT INPUT B
40  MAT A = B / (10)
50  MAT PRINT A

RUN
? 10 20 30 40
? 100 200 300 400

1                    2                    3                    4
A                    14                   1E                   28
```

9.9 Identity matrix

General form : MAT Arr1 = IDN (xp1, xp2) where :

.Arr1 is the name of a two-dimensional numeric matrix.
.xp1 and xp2 are arithmetic expressions.

Purpose : To establish Arr1 as an identity matrix and optionally specify a new working size.

Comments : .An identity matrix is one having all its elements set to zero, except those residing on its diagonal from upper left to lower right which are set to one.

.If a new working size is not specified, the original matrix must be square (the number of rows must equal the number of columns, after re-dimensioning if any).

.The clause "(xp1, xp2)" is optional. If present, it specifies a new working size for Arr1, and therefore has the same effect as a DIM, BYTE or INTEGER statement . xp1 and xp2 must obey the same rules as those followed by dimension sizes (see paragraph 3.5.2). In addition, the number of elements implied by (xp1, xp2) must not exceed the number of elements the matrix Arr1 has prior to execution of the statement. Furthermore, in the case of the matrix identity statement, xp1 must equal xp2.

Example 9.8 : Matrix identity.

```
10  DIM A(30,20),B(2,2)
20  MAT A=IDN(3,3)
30  MAT B=IDN
40  MAT PRINT A
50  MAT PRINT B

RUN

1                  0                  0
0                  1                  0     ( matrix A )
0                  0                  1

1                  0                        ( matrix B )
0                  1
```

9.10 The MAT SET statement

General form : MAT Arr1 = SET (xp1, xp2) [exp] where :

.Arr1 is the name of a one- or two-dimensional numeric array.
.xp1, xp2, and exp are arithmetic expressions.

Purpose : To set all the elements of an array Arr1 to the value specified by "exp", and to optionally establish a new working size as indicated by (xp1, xp2).

Comments : .The new working size specification is optional; the rules given under 9.9 apply, except that xp1 and xp2 need not be identical.

Example 9.9 : Erasing (filling with blanks) a video RAM based at hex address $E000.

```
10  BYTE SCREEN(22,80) ADDRESS $E000
20  MAT SCREEN = SET [$20 ] \ or SET [ASC(" ")]
```

9.11 The MAT ZER and MAT CON statements

These two statements are particular cases of the MAT SET statement, in that they allow to fill an array with two specific values : 0 (MAT ZER), or 1 (MAT CON). The syntax is as follows :

General form : MAT Arr1 = ZER (xp1, xp2)
MAT Arr1 = CON (xp1, xp2)

where Arr1, xp1, xp2 have the same meaning as in the MAT SET statement.

9.12 Matrix transposition

General form : MAT Arr1 = TRN (Arr2)

where :

.Arr1 and Arr2 are names of two-dimensional numeric arrays.

Purpose : To replace the elements of an array Arr1 with the matrix transpose of another array Arr2.

Comments : .The number of elements in Arr2 must not exceed the number of elements in Arr1.

.The values in column y of Arr2 become the values in row y of Arr1.

.The dimensions of the resulting matrix Arr1 are set to be the reverse of the original matrix Arr2. For instance, if A has dimensions of (4,2) and MAT B=TRN(A), B will be assigned the dimensions of (2,4).

.Matrices cannot be transposed into themselves.

Example 9.10 : Matrix transposition.

```
10  DIM A(50,2),B(2,3)
20  MAT INPUT B
30  MAT A = TRN(B)
40  MAT PRINT A

RUN
? 1 2 3
? 4 5 6

1                   4      ( from now on , A assumes
2                   5        the dimensions of (3,2) )
3                   6
```

9.13 Matrix inversion

General form : MAT Arr1 = INV (Arr2)

where :

.Arr2 is the name of a two-dimensional square numeric array.
.Arr1 is also a two-dimensional numeric array to be dimensioned as Arr2 after inversion.

Purpose : To establish a square matrix Arr1 as the inverse of a specified square matrix Arr2.

Comments : .For the square matrix Arr2 of dimensions (m,m), the inverse matrix Arr1, if it exists, is a matrix of identical dimensions such that:

Arr1*Arr2 = Arr2*Arr1 = I, where

I is an identity matrix.

.Not every matrix has an inverse: the inverse of a matrix Arr2 exists if its determinant is different from zero.

.When the determinant is equal to zero, an error occurs. In some cases, the determinant is very close to zero, thus causing either error conditions, or yielding a result which is far from the true inverse.

Example 9.11 : Solving a system of linear equations.

Giving the following :

```
all.xl + ..... + aln.xn = bl
a2l.xl + ..... + a2n.xn = b2
               :
anl.xl + ..... + ann.xn = bn
```

the values xl, ... , xn which satisfy all the equations are such that :

X = INV(A) * B

where :

X is the resulting vector { xl, ..., xn },
A is the square matrix containing the coefficients aij,
B is the vector { bl, ..., bn }

```
10  DIM A(4,4),B(4),X(4)
20  PRINT "enter aij "
30  MAT INPUT A
40  MAT INPUT "enter bi ", B
50  MAT A = INV(A)
60  MAT X = A*B
70  MAT PRINT X
```

```
RUN
enter aij
? 1 5 8 7
? 9 3 6 8
? 2 7 4 9
? 9 1 3 2
enter bi ? 10 3 6 2

-0.130643612         1.34005764          1.19788665          -0.878962535
```

CHAPTER 10

10. FUNCTIONS AND PROCEDURES

Much of the art of programming lies in recognizing problems which can be solved by a repetitive sequence of operations. Repetition is mainly desired because it enhances the effectiveness of a given number of instructions, results in a shorter code, and makes for economy of thought on the part of the programmer.

Two kinds of repetitions are found in programming, as illustrated in the following time sequences where "ti" stands for task i :

(1) t1 t2 t2 t2 t3 ......

(2) t1 t2 t3 t2 t4 t2 .....

The connected repetition in (1) would be written as a loop; task "t2" would then be written once only in the source program, and the corresponding instructions coded once only in the object program.

Disconnected repetitions, as in (2), can be solved by using subroutine calls, a subroutine being a sequence of instructions, written and stored once only. In BASIC-M, subroutines are called by the GOSUB statement. There are two main drawbacks in using GOSUB´s:

1/ Programs are not very readable because the subroutine is not represented by a label which would help to guess its function, but rather by a line number.
2/ Arguments or parameters cannot be passed easily to the subroutine.

In BASIC-M, these problems can be overcome by using procedures and functions:

- A procedure is a sequence of instructions, thus represented on several statement lines, which is executed whenever the procedure name is encountered in the program. Arguments can be associated with procedures, and passed back and forth between the calling program and the procedure.

- A function is a type of subroutine which returns a single result to the calling program. Two types of functions exist:

.User-defined functions declared on a single statement line.

.Built-in BASIC-M functions such as sine, substring search, or logarithms.

The rest of this chapter details the user-defined procedures and functions, while chapter 11 discusses the built-in functions.

10.1 User-defined functions

Such functions are useful when a particular numeric or literal expression appears many times in a program. They are declared by using the DEF statement whose form is shown next.

General form : DEF funct(arg1, arg2, ...) = exp

where :

.funct , arg1, arg2 are the names of the function and formal arguments respectively; these names conform to the rules followed by the BASIC-M variable names.
.exp is an arithmetic or literal expression, depending on whether the function is of the numeric or character type.

Purpose : The DEF statement is used to define a user function and its associated formal parameters, if any.

Comments : .The DEF statement defining a function must appear before any use of the function.

.If the function name "func" ends with a dollar sign "$", the function is of the character type; consequently, the expression on the right-hand side of "=" must be literal.

.If the function name does not end with the dollar sign, the function is numeric and assumes the type real; the expression must then be an arithmetic expression (its components may be of any numeric type, not necessarily real).

.A function must not have the name of a variable or procedure.

.A user-defined function can be used wherever an expression is allowed by writing:

...... func(arg1, arg2, ..., argN) ......

.When the function is invoked, the actual arguments (those stated in the function call) are used to evaluate the expression that yields the function result. As an example, let's consider a function "Sum(X,Y)" which is supposed to return the sum of two parameters X and Y. The following listing shows how the function is defined, then further invoked to print the sum of two input values A and B.

```
10  DEF Sum(X,Y) = X+Y
20  INPUT A,B
30  PRINT Sum(A,B)
40  IF A # 0 THEN 20
50  STOP
```

X and Y are the formal parameters of the user-defined function "Sum", while A and B are the actual parameters.
When the function "Sum" is invoked in line 30, the following action is taken by the runtime package :

-The values in A and B, are converted to the type of X, respectively Y, if the types of the actual arguments do not match those of the formal parameters.
-These values, once converted, undergo the computation dictated by the function definition (line 10), where the value of A is equivalent to the value of X, and the value of B is equivalent to the value of Y.

The conversion, which may occur when the function is invoked, can be illustrated on the same example, in which an additional statement line would declare Y as an integer (5 INTEGER Y). The following results would be obtained:

```
RUN
? 2 9
11
? 2 32769
-32765
```

At first glance, the second set of inputs does not produce the expected result; this is due to the fact that Y has been declared as an integer; because integers are 16-bit signed quantities, 32769 corresponds to -32767, hence the final result. It is recommended to insure type compatibility between the actual and the formal arguments.

The number of arguments in a function call also calls for some comments. If the number of actual arguments in the call is less than the number of formal arguments in the definition, the extra arguments appearing in the expression of the definition assume the value they were given prior to the function call. The following is an example:

```
10  D?EF Sum(X,Y)=X+Y
20  INPUT A,B
       :
50  Y=1
       :
80  PRINT Sum(A)
       :
RUN
? 100 200
101
?
```

If the number of arguments in the call is greater than the number of formal arguments in the definition, the extra actual arguments are merely ignored. This is visualized on the following:

```
10  DEF Sum(X,Y)=X+Y
20  INPUT A,B,C,D
30  PRINT Sum(A,B,C,D)

RUN
? 1 10 100 1000
11
```

The actual arguments arg1, arg2,..., argN involved in a function call may be any valid expressions, and therefore may consist of simple variables, array elements, ... or user-defined functions. Complete arrays, however, cannot be passed as arguments. Below is another example of function calls.

```
1   REM Sum the elements of a matrix A
10  DIM A(4)
20  MAT INPUT A
30  DEF Sum(X,Y)=X+Y
40  PRINT Sum(Sum(Sum(A(1),A(2)),A(3)),A(4))
50  REM same as .....
60  PRINT Sum(A(1),A(2)+A(3)+A(4))

RUN
? 1 2 3 4
10
10
```

Example 10.1 : Compute the roots of a quadratic equation.

```
10  DEF DET(R,S,T)=S*S-4*R*T    \ function to return determinant
20  INPUT A,B,C                 \ enter coefficients of equation
30  A$="[C-3(.)2] +- ([C3(.)2])" \ define format string
40  D=DE-T(A,B,C)                \ call function
50  IF D>=0 THEN 80
```

```
60   A$=A$+"*i"                      \ if determinant is < 0, then
70   D=-D                        \ negate , and modify format string
80   PRINT USING A$,-B/(2*A),SQR(D)/(2*A)            \ print roots
90   GOTO 20

RUN
? 9 3 4                        ( 9*X*X + 3*X + 4 )
-000.17 +- (000.65)*i
? -10 4 6                      (-10*X*X + 4*X + 6 )
 000.20 +- (000.80)
?
```

Example 10.2 : Defining a string function.

```
10   DEF QUEST(A$) = A$+"?"   \ function to append a
20   INPUT X$                 \ question mark to a string
30   PRINT QUEST(X$)
40   PRINT QUEST(QUEST(QUEST(X$)))         \ append 3 "?"

RUN
? "DOES IT WORK "
DOES IT WORK ?
DOES IT WORK ???
```

10.2 Procedures

Procedures are defined using a different form of the DEF statement.

General form : DEF proc(arg1, arg2, ...)

where :

.proc, arg1, arg2 are the names of the procedure and formal arguments respectively, which conform to the rules followed by BASIC-M variable names. A procedure must not be given the name of a variable or function.

Purpose : The DEF statement is used to define a series of statements which may be later invoked by writing the name of the procedure, possibly followed by an argument list. Alternatively, the CALL statement can be used to invoke the procedure execution; the syntax of the CALL statement is as follows:

CALL proc (arg1, arg2 ,...)

Comments : .The procedure is exit on execution of the first balancing RETURN statement which is encountered.

.The procedure definition must appear before any use of the procedure.

.A procedure does not assume any type since no result is returned to the calling program.

.The rules applying to the arguments of a user-defined function apply also to the arguments of a procedure (see paragraph 10.1).

.The programmer must make sure that program control is never transferred directly to the procedure (unless carely planned), as would be the case with the following program structure:

```
        :
110  A=1
120  DEF DELAY(X)
        :
150  RETURN
160  PRINT A
        :
```

The execution of the RETURN statement at line 150 would cause the stack to be updated without any valid reason (no procedure call was made before). This would result in a runtime fatal error. To avoid this situation, it is a good programming practice to precede a procedure definition by a GOTO statement so as to skip the procedure body in case of in-line execution. The compiler issues a warning message whenever a procedure (not a user-function !) definition is not preceded by a GOTO statement. A warning message does not prevent the program from being executable. These two detections (at compile and run time) are illustrated below:

```
10  A=1
20  PRINT A
30  DEF DELAY(X)
40  FOR K=1 TO X
50  NEXT K
60  RETURN
70  PRINT 2*A
```

```
RUN
*** WARNING *** PROCEDURE LINE 00030 (compilation)
1                                    ( execution )

*** FATAL ERROR #133 AT LINE    60
```

Example 10.3 : Procedure to complement the bit #i of a byte (bit #0 is the rightmost bit).

```
10   BYTE BYT, PIA ADDR $8008
20   GOTO 100          \ skip procedure definition
30   DEF COMP(BYT,I)       \ .. procedure definition ....
40   BYT[I]=IEOR(BYT[I],1)
50   RETURN                \ .. procedure physical end ..
90   REM ---- MAIN ----
100  INPUT K
110  COMP(PIA,K)       \ invert bit #K of PIA
120  IF PIA = 0 THEN STOP
          :
```

Example 10.4 : Exchange two elements of a string vector.

```
10   DIM A$(3)
20   GOTO 100
30   DEF EXG(I,J)         \ procedure definition
40   T$=A$(I)             \          :
50   A$(I)=A$(J)          \ procedure body
60   A$(J)=T$             \          :
70   RETURN               \ physical end
80   REM ---- MAIN ----
100  MAT INPUT A$
110  MAT PRINT A$
120  EXG(1,2)
130  MAT PRINT A$

RUN
? "DO ","YOU ","UNDERSTAND"

DO YOU UNDERSTAND       ( due to line 110 )

YOU DO UNDERSTAND       ( due to line 130 )
```

Example 10.5 : .The sample program below makes use of 3 procedures to set a bit in a byte, to store the byte in a buffer when all eight bits have been set, and finally to output the buffer once full.

This example is intended to demonstrate the nesting of procedures.

```
10   BYTE BUF(3),II
20   GOTO 500
30   REM
100  REM ------ PUT_REC --------
110  DEF PUT_REC            \ when buffer is full, then
120  MAT PRINT BUF          \ print it and ....
130  P=0                    \ reset pointer
140  RETURN
150  REM ------ PUT_BYTE -------
200  DEF PUT_BYTE(I)
210  INDEX=(P+7)/8
220  BUF(INDEX)=I          \ store byte according to
230  IF INDEX=3 THEN PUT_REC   \  value of pointer
```

```
240 RETURN
250 REM ------ PUT_BIT --------
300 DEF PUT_BIT(J)      \ set bit currently pointed
310 II[7-IAND(P,7)]=J   \ to by P to the value J
320 P=P+1               \ update pointer, and store
330 IF IAND(P,7)=0 THEN PUTBYTE(II) \ full byte if
340 RETURN                          \ appropriate.
350 REM
360 REM ***** MAIN PROGRAM ********
500 P=0                 \ initialize pointer.
       :
730 PUTBIT(K)
  :
```

10.3 Assembly language interface

In some cases, to speed up the execution of the overall program, it is more efficient to code certain tasks in assembly language. This section discusses how to link these portions of codes to a BASIC-M program.

The assembly language programs must obey the following rules:

- they must be structured as subroutines, and therefore should terminate with an "RTS" instruction or the like.
- they must be declared as "external" subroutines (refer to 7.4).

Depending on the context (the way they are invoked), the assembly language subroutines will be used as procedures or as user-defined functions (these latter only are supposed to return a value onto the MC6809 User Stack.).

Context : .When an external subroutine is called like a procedure, i.e, by just writing its name optionally preceded by the CALL keyword, it executes exactly as a procedure does. The subroutine exits upon execution of the "RTS" or equivalent instruction. The following are valid examples of assembly language procedure calls.

```
10   EXTERNAL ERASE ADDR $D400, Switch $D430
        :
35   IF SCREEN(22,80) #$20 THEN ERASE
        :
60   CALL Switch(X)
        :
85   Switch
```

.When an external subroutine is referenced in an expression, it is used as a user-defined function, and therefore must return a single result on the user stack. When used in this context, assembly routines whose name ends with a "$" are expected to return a string result (32 bytes), whereas the others are supposed to return a real data onto the user stack (5 bytes). The format of the returned data must agree with the BASIC-M internal representation of the character or real variables (refer to chapter 3).

Arguments : .When arguments are associated with an external subroutine call, the subroutine is entered with the MC6809 Y-register pointing to a table that contains the addresses of the arguments; this table is structured as follows:

```
! FDB Arg1 ! <----- Y-REG
! FDB Arg2 !
 !     :     !
! FDB ArgN !
! FDB 0    !
```

where each line stores a 16-bit address, with the last one being zero (terminator).

.The actual arguments may be simple variables, or expressions. The user is responsible for insuring type compatibility between the BASIC-M variables and his assembly language variables: in other words, BASIC-M byte variables must be handled as bytes, integer as 16-bit words, ... etc.

Example 10.6 : Defining an assembly subroutine to erase the EXORset display (alphanumeric memory based at hex address $E000).

```
10  BYTE Screen(22,80) ADDRESS $E000
20  EXTERNAL ERASE ADDRESS $D400
30  ERASE
       :
```

```
Assembly subroutine
-------------------
          ORG $D400

SCREEN    EQU  $E000      Display base address

ERASE     LDX  #SCREEN
          LDD  #$2020     Two blanks
ERS       STD  ,X++
          CMPX #SCREEN+2048
          BNE  ERS        Go on erasing
          RTS
```

Example 10.7 : Change all upper-case characters of a string to lower-case. This is an example of an external assembly language user function call with a literal expression as argument.

```
10  EXT LOWER$ ADRRESS $D500
20  INPUT A$,B$
30  PRINT LOWER$(A$+B$)
40  GOTO 10

RUN
? "BASIC-M ","User´s Guide"
basic-m user´s guide
?
```

Assembly language function

```
                         ORG $D500

33 C8 E0  LOW      LEAU   -32,U      create room for string
6F C4              CLR    ,U         default to empty string
AE A4              LDX    ,Y         get string address
27 13              BEQ    EXIT       no argument in call
86 1F              LDA    #31
E6 86     LOOP     LDB    A,X        transfer string to stack
C1 41              CMPB   #´A        upper-case char ?
25 06              BLO    IGNOR
C1 5A              CMPB   #´Z        (the string length byte is
24 02              BHI    IGNOR       not an upper-case char code)
CA 20              ORB    #$20       yes. change to lower-case
E7 C6     IGNOR    STB    A,U        store in result string
4A                 DECA              tally counter
2A EF              BPL    LOOP
39        EXIT     RTS               All done. Return.
```

CHAPTER 11

11. BUILT-IN FUNCTIONS

This chapter covers the functions defined in BASIC-M, which fall into five categories. The first part of the chapter lists and describes these while the second part contains several application examples.

11.1 Trigonometric functions

SIN(X)	sine of X radians.
COS(X)	cosine of X radians.
TAN(X)	tangent of X radians.
ATN(X)	arctangent of X. result in radians. -PI/2 < result < +PI/2.
ATN(X,Y)	ATN(X/Y).
ASN(X)	arcsine of X. result in radians.
ACS(X)	arcosine of X. result in radians.
SINH(X)	hyperbolic sine of X.
COSH(X)	hyperbolic cosine of X.
TANH(X)	hyperbolic tangent of X.
COTH(X)	hyperbolic cotangent of X.

11.2 Other mathematical functions

EXP(X)	natural exponent of X.
LOG(X)	logarithm of X to the base e.
DCLOG(X)	logarithm of X to the base 10.
SQ(X)	square of X (X*X).
SQR(X)	square root of X.
ABS(X)	absolute value of X.
SGN(X)	sign of X. SGN(X)=-1 if X<0. SGN(X)=0 if X=0.

	SGN(X)=1 if X>0.
SGN(X,Y)	SGN(X,Y) = ABS(X) * SGN(Y).
INT(X)	truncate value of X to an integer.
MOD(X,Y)	modulus. returns the remainder of the division of X by Y.
RND	generates a pseudo random number between 0 and 1. A new random number is produced each time the "RND" function is invoked within the program; the sequence of random numbers using "RND" is identical each time a program is run.
RND(X)	initializes the congruential series to the value of X and returns a random number. Because random numbers generation depends on the initial value of the series, RND(X1)=RND(X2) if X1=X2. X must be in the range 0,1. Note that RND(0) is equivalent to RND.
FIX(X)	returns the 2-byte integer corresponding to the real value X (-32768 <=X<= 32767).
FLOAT(X)	returns the 5-byte real corresponding to the integer value X.

11.3 Logical functions

IAND(X,Y)	logical AND of X and Y.
IOR(X,Y)	logical inclusive OR of X and Y.
IEOR(X,Y)	logical exclusive OR of X and Y.
ISHFT(X,Y)	shifts the binary value of X by Y positions to the left if Y>0 or to the right if Y<0.

11.4 String functions

LEN(X$)	returns the length of X$.
LEFT$(X$,Y)	returns the leftmost Y characters of the string X$.
RIGHT$(X$,Y)	returns the rightmost Y characters of the string X$.
MID$(X$,Y,Z)	extracts a string from the string X$, which begins Y positions from the left and continues for Z characters.
TRIM$(X$)	removes trailing blanks from X$.

ASC(X$)	returns the numeric value of the code of the first ASCII character within a string X$.
CHR$(X)	returns a single character whose ASCII code is equivalent to the value of X.
STR$(X)	returns the ASCII form of the numeric value X as if it were printed.
STR$(X,Y$)	same as STR$(X) except that the format directives are taken from the string Y$.
VAL(X$)	returns the numeric constant equivalent to the numeric string X$. Thus VAL("4E3")=4000.
SUBSTR(X$,Y$)	returns the position of substring Y$ in string X$. If substring not found, returns 0.

11.5 Miscellaneous functions

PEEK(X)	returns the byte stored at location X.
POKE(X,Y)	stores the byte value of Y at location X. Note that this is not a function, in that it does not return any value to the calling program. It is listed here as this statement performs the reverse operation of the PEEK function.
LOC(Arg)	returns the address of Arg. Arg must be a simple variable or subscripted variable name.
TAB(X)	causes tabulation to column X. Only allowed in a PRINT statement.
POS	returns the current position of the cursor or print head of the console device. The POS value is updated only each time the print buffer is flushed.
ERR	returns the code of the error that occured last (see chapter 8). This is a read-modify-write function.
FKEY	returns the code of the function key which was hit last (see chapter 8). This is a read-modify-write function.
EOF(X)	set to one when end of file X is reached (see chapter 12). When X has the value 0, EOF allows to check if the pointer associated with the READ and DATA statements is pointing to the end of the data table (see paragraph 5.3); if yes, EOF returns a one.

11.6 Default type of the arguments

As already mentioned, BASIC-M takes care of the conversion of the arguments involved in a function call to the type of the formal arguments. These conversions, when necessary, consume some of the overall program execution time. When speed is at a premium, it is of benefit that the types of the actual arguments match those of the formal arguments. The type of the formal arguments involved in the BASIC-M built-in functions is shown in the next table.

function	Arg1	Arg2	Arg3	Result
SIN(X)	R	-	-	R
COS(X)	R	-	-	R
TAN(X)	R	-	-	R
ATN(X,Y)	R	R	-	R
ASN(X)	R	-	-	R
ACS(X)	R	-	-	R
SINH(X)	R	-	-	R
COSH(X)	R	-	-	R
TANH(X)	R	-	-	R
COTH(X)	R	-	-	R
EXP(X)	R	-	-	R
LOG(X)	R	-	-	R
DCLOG(X)	R	-	-	R
SQ(X)	R	-	-	R
SQR(X)	R	-	-	R
ABS(X)	R	-	-	R
SGN(X,Y)	R	R	-	R
INT(X)	R	-	-	R
MOD(X,Y)	R	R	-	R
RND(X)	R	-	-	R
FIX(X)	R	-	-	I
FLOAT(X)	I	-	-	R
IAND(X,Y)	I	I	-	I
IOR(X,Y)	I	I	-	I
IEOR(X,Y)	I	I	-	I
ISHFT(X,Y)	I	I	-	I
LEN(X$)	S	-	-	B
LEFT$(X$,Y)	S	B	-	S
RIGHT$(X$,Y)	S	B	-	S
MID$(X$,Y,Z)	S	B	B	S
TRIM$(X$)	S	-	-	S
ASC(X$)	S	-	-	B
CHR$(X)	B	-	-	S
STR$(X,Y$)	R	S	-	S
VAL(X$)	S	-	-	R
SUBSTR(X$,Y$)	S	S	-	B
PEEK(X)	I	-	-	B
POKE(X,Y)	I	B	-	-
LOC(X)	-	-	-	R
TAB(X)	B	-	-	-
POS	-	-	-	B
ERR	-	-	-	B
FKEY	-	-	-	B
EOF(X)	B	-	-	B

B = byte I = integer R = real S = string

11.7 Illustrative examples

Example 11.1 : Using the RND function.

```
10   PRINT RND, RND          \2 different numbers
20   REM         are produced because no argument
30   PRINT RND(.2),RND, RND(.2)

40   REM "RND(.2)" always yields the same number

RUN
1.39698386E-09        9.15583223E-05
0.199987794           7.32536428E-05       0.199987794
```

Example 11.2 : Display a date in the form
Date : MM/DD/YY
on the bottom line of the EXORset screen.

```
10   DIM DATE$(1) ADDRESS $E6B0
20   BYTE Dummy ADDR DATE$
30   INPUT Date
40   DATE$=STR$(Date,"Date : [C2/2/2]")
50   Dummy=ASC(" ")
60   GOTO 30

RUN
? 70479
?                                Date : 07/04/79
```

Note : line 50 blanks the location corresponding to the byte that contains the length of the string DATE$.

Example 11.3 : Count the number of characters "M" in an input string A$.

```
10   INPUT A$                    \ input string
20   K=0                         \ default to no "M"
30   Posit = SUBSTR(A$,"M")      \ search for next "M"
40   IF Posit=0 THEN 80          \ none. exit.
50   K=K+1                       \ another one. update counter.
60   A$=RIGHT$(A$,LEN(A$)-Posit) \ shrink string.
70   GOTO 30                     \ go on searching for next "M"
80   PRINT K                     \ print amount of "M" found
90   GOTO 10

RUN
? BASIC-M MANUAL
2
?
```

Example 11.4 : Using "POS" to print column numbers.

```
10  LINE=60
20  PRINT
30  PRINT "          ";
40  PRINT USING "[1]",POS/10;
50  IF POS # 1 THEN 30
60  PRINT USING "[1]",MOD(POS,10);
70  IF POS # 1 THEN 60

RUN

         1         2         3         4
1234567890123456789012345678901234567890123456789012 ........
```

(partial listing of the result)

CHAPTER 12

12. DISK FILE INPUT/OUTPUT

When running in an XDOS/MDOS environment, a BASIC-M program may transfer data to or from disk files. This chapter describes the disk I/O statements and the file interface between BASIC-M and XDOS/MDOS floppy disk operating system.

12.1 General description

BASIC-M file management package uses some XDOS/MDOS routines to interface with disks, that means file input/output is allowed only within a program running under XDOS/MDOS control. The user is recommended to be familiar with the file and disk structures described in the MDOS III User´s Guide or the XDOS User´s Guide prior to manipulating files with BASIC-M.

Since BASIC-M performs all its I/O transfers through logical units, a number has to be assigned to each file when it is opened: this number must be a positive integer, not greater than 255 and may not be a standard peripheral logical unit number.

There are three file access types available with BASIC-M: sequential, random and indexed. These will be discussed in paragraph 12.1.1.

Also, there are three open modes: for input only, for output only, for both input and output (update).

IMPORTANT WARNING

SINCE THE DISK FILE INPUT/OUTPUT PACKAGE AND THE DISK OPERATING SYSTEM ARE NOT REENTRANT, A PROGRAM WHICH MANIPULATES DISK FILES MUST NOT BE INTERRUPTED. INTERRUPTING A DISK FILE I/O INSTRUCTION MAY DESTROY THE DISK DATA AND FILE STRUCTURE. DO NOT USE "ON KEY", "ON NMI", "ON IRQ", "ON FIRQ" AND DISK I/O SIMULTANEOUSLY.

12.1.1 File types

Three file types are available :

sequential — the file is read or written one record after another, beginning with the first one. No positioning is allowed. This type follows the ASCII source record structure described in the XDOS user´s guide or the MDOS III User´s Guide.

random — a random file is made of fixed length records, each of them may be addressed by its ordinal position in the file, beginning with record 1. Each record physically exists, even if it has not been written into it, except those records following the last record written in the file. This means that any unused record between existing records occupies the same amount of room in the file as if it were initialized.

indexed — as a random file, an indexed file is made of fixed length records. A table follows the last data record in the file: it contains numbers which are used as record access keys. An index key is not the position of the record in the file, but only a number assigned to it. Since records are entered sequentially in the file when a new key is used, no "holes" appear between existing records. For example, an indexed file in which only records with index keys 4 and 2097 have been written, contains effectively two records.

Use of random and indexed files is very similar; the choice between these two file structures is a matter of access speed and mass storage occupation: a random file requires more room than an indexed file if the records used are not contiguous, but the access is much faster, especially when the file contains a lot of records. The user has then to choose the most convenient file structure, depending on its own application requirements.

For listing purposes, any type of file is created as an XDOS/MDOS ASCII file, however, the physical organization slightly differs from one file type to another.

	sequential	indexed	random
record length	variable	fixed	fixed
maximum record length	255 bytes	255 bytes	255 bytes
space compression used	yes	no	no
RIB entries (see XDOS/MDOS User´s Guide)			
RIB$LB	0	1	2
RIB$SL	0	number of data sectors in file.	0
RIB$SA	0	record length.	record length.
RIB$LA	0	MS byte : number of sectors in index table.	0

See XDOS/MDOS User´s Guide for file structure and RIB descriptions.

From this table, we see that a sequential file is completely compatible with XDOS/MDOS software, whereas the random and indexed files are not standard because special entries in RIB are used.

Care should therefore be exercised when manipulating random and indexed files with XDOS/MDOS subsystems or other non-BASIC-M programs:

- Copying one of these files will destroy the pointers in the destination file.
- MDOS REPAIR command will find discrepancies in RIB and will generate a warning message.
- Garbage records will appear at the end of an indexed file listing ; they are due to the output of the index table.

File accesses

File accesses refer to the operations performed for transferring data to or from a record in the file. The access of a specific file is declared in an OPEN statement (see paragraph 12.2).

For each opened file, there is a pointer. This pointer indicates the file record on which the next data transfer will be performed. After complete transfer of a record, the pointer points to the next record, which allows consecutive record transfer through execution of a program.

This pointer may be modified at each data transfer request (INPUT AT, PRINT AT) or by a REWIND statement (see paragraph 12.6): this allows the user to transfer data to or from a specific record without reading or writing the file completely. Use of pointer positioning clauses depends of the access in use within the file:

Sequential — The positioning requests issued within data transfer statements are ignored. The user may only REWIND the file.

Random — Positioning requests within data transfer statements are optional: If not specified, transfers occur sequentially, else, the integer value provided by the request is considered as the position in the file of the record to be accessed. Subsequent accesses without positioning request are performed on the consecutive records. The first record in the file is in position 1. REWINDing a random file will cause the next data transfer to occur on record position 1, unless there is a positioning request issued by the data transfer statement.

Indexed — The positioning clause in data transfer statements is compulsory. The value provided by this clause is the index key, and will be searched in the index table. Index key may be any integer value between -32768 and +32767. If the positioning request clause is omitted, the data transfer will be performed on the record whose index key is 0. A REWIND statement has no effect when issued on a file open for indexed access.

Any of the three file types may be opened for sequential access, while only random files may be open for random access and only indexed files for indexed access.

When a record is not terminated (the previous file access statement was a PRINT and the last delimiter was a comma or a semi-colon), the only statement which will continue the same record is a PRINT without pointer positioning clause. A record may never be continued in an indexed file.

12.2 The OPEN statement

This statement is used to open a file and assign it to a logical unit number.

General form : OPEN #LU,filename,mode,access,recl

where :

LU is an expression representing the logical unit number to be assigned to the file. After evaluation, it is truncated to its integer value. This must be a valid disk logical unit number (3 <= LU <= 255).

filename is a string variable or constant containing the name of the file to be opened. It must be a valid XDOS/MDOS file name. The default suffix is "SA", the default drive is 0 (See XDOS/MDOS User´s Guide for the complete file name description).

mode is the transfer mode used within the file. It may be

I	for input
O	for output
U	for update (input and output).

access this parameter specifies the access type in use at each data transfer. This can be :

SEQ	for sequential access
RAN	for random access
IND	for indexed access

This parameter is optional; if not provided, SEQ is assumed.

recl is an optional integer constant representing the record length in the file. This parameter is not used if the file to be opened already exists or if the file is accessed sequentially. If not provided, the global line length parameter value is used (see paragraph 5.6).

Depending on mode, access and file existence, OPEN performs different operations :

file exists	sequential	indexed	random
input mode	open for input, point to the first record in the file.	open for input, no file pointer positioning.	open for input, point to the first record in the file.
output mode	fatal error.	fatal error.	fatal error.
update mode	open for I/O, point to the end of file.	open for I/O, no file pointer positioning.	open for I/O, point to the first record in the file.

new file	sequential	indexed	random
input mode	file is not created, end of file flag is set, file may only be tested (see paragraph 12.4) or closed.		
output and update mode	create and open file.	create and open file, initialize RIB values, build an empty index table.	create and open file, initialize RIB values.

A maximum of five files may be open at the same time. If more files are needed, an opened file must be closed before opening another one.

12.3 The file INPUT statement

This statement is used to position the file pointer and accept data from the file.

General form : INPUT #LU AT key,var1,var2,...,varn

where :

LU is an expression whose result is translatable to a byte value, representing the logical unit

	number assigned to the file upon which data transfer will be performed.
key	is an expression translatable to an integer value which will be used as the index key. The "AT key" clause is optional. See paragraph 12.1.2 for details about file pointer positioning.
var1,var2,varn	are variables into which the input values are to be stored. If there are more values in the record than the number of variables in the list, the excess values are ignored. If there are not enough values in the record to fill all variables, the last variables in the list are unchanged. This allows variable initializing with a default value before input. On the other hand, it constraints the user to clear the variable before each input if the previous variable content is not to be considered as a default value.

An illegal logical unit number causes an error message to be printed and data to be accepted from the console keyboard.

Positioning the file pointer (by the AT clause) with a valid index key always resets the end of file flag.

The end of file flag is set upon statement completion if :

- The end of a sequential file has been read.
- A record located beyond the last record of a random file has been read.
- An index key which is not yet entered in the table (uninitialized record) has been given for the positioning of an indexed file pointer.

After an end of file condition has occured, consecutive read without positioning the file pointer will cause a fatal error.

Care should be taken if reading of a sequential file opened in update mode is needed : the file pointer is at end of file. To read the existing records in file, a REWIND statement (see paragraph 12.6) must occur prior to the first INPUT statement execution.

Data input from an uninitialized random file record is not detected and the data read are unsignificant, since the file is not zero filled at creation time.

Example 12.1: The following program interprets operation codes located in a random file considered as a "virtual memory" (the program which creates the file is not shown). Each record is 40 bytes long and contains at most two numbers: the first one is the operation code or data, the second one is an optional parameter representing the record number upon which the operation must be performed. Operation codes are defined as follows.

op-code	function
1	Jump to record specified by the parameter.
2	Load the first number contained in the record pointed by the parameter into the accumulator.
3	Add to the accumulator the first number contained in the record pointed by the parameter
4	Multiply the accumulator by the first number contained in the record pointed by the parameter.
5	Store the accumulator content in the record pointed by the parameter.
6	Input a number from the keyboard and store it in the record pointed by the parameter.
7	Print the first number contained in the record pointed by the parameter.
8	Stop the execution.
9	Test the first number of the record pointed by the parameter: if zero, do not execute the next operation code.

```
10   OPEN #4,"MEMORY",U, RAN    \ open virtual memory file
20   PC=1             \ initialize program location counter
30   ACC=0                              \ reset accumulator
40   INPUT #4 AT PC ,OPCODE,MEM \ fetch op-code & operand
50   IF OPCODE<1 OR OPCODE>9 THEN 390    \ illegal op-code
60   PC=PC+1               \ point to next program location
70   ON OPCODE GO TO 80,110,140,180,220,250,290,330,350
80   REM -JUMP-    OPCODE = 1
90   PC=MEM
100  GO  TO 40
110  REM -LOAD ACCUMULATOR-    OPCODE = 2
120  INPUT #4 AT MEM ,ACC              \ MEM is value address
130  GO  TO 40
140  REM -ADD MEMORY TO ACCUMULATOR-    OPCODE = 3
150  INPUT #4 AT MEM ,DUMMY
160  ACC=ACC+DUMMY
170  GO  TO 40
180  REM -MULTIPLY ACCUMULATOR BY MEMORY-  OPCODE = 4
190  INPUT #4 AT MEM ,DUMMY
200  ACC=ACC*DUMMY
210  GO  TO 40
220  REM -STORE ACCUMULATOR IN MEMORY-  OPCODE = 5
230  PRINT #4 AT MEM ACC
240  GO  TO 40
250  REM -INPUT A VALUE AND STORE IN MEMORY-  OPCODE = 6
```

```
260  INPUT DUMMY
270  PRINT #4 AT MEM DUMMY
280  GO  TO 40
290  REM -PRINT MEMORY CONTENT-  OPCODE = 7
300  INPUT #4 AT MEM ,DUMMY
310  PRINT DUMMY
320  GO  TO 40
330  REM -STOP-  OPCODE = 8
340  STOP " ****  END OF RUN  ****"
350  REM -SKIP IF ZERO-   OPCODE = 9
360  INPUT #4 AT MEM,DUMMY
370  IF DUMMY = 0 THEN PC = PC + 1
380  GO TO 40
390  REM
400  STOP "****    ILLEGAL OP CODE    ****"
410  END
```

12.4 The end of file test

There are two non-standard forms of test statements used to take special actions upon reading the end of a file. These are :

IF EOF(LU) THEN action
WHEN EOF(LU) THEN action

where :

LU	is an expression representing the logical unit number on which the file to be tested is open (see paragraph 12.3).
action	is a line number or an executable statement which does not include a THEN clause (exceptions : FOR, NEXT). See paragraph 5.11 for a complete description.

The EOF function value is true if the end of file flag of the requested logical unit is set. That means the action will be taken if the end of the file has been encountered or an unknown index key has been used in the last file access.

Immediately after opening a file in input mode, the function result is true if the file does not exist.

In addition, if the last DATA item (see paragraph 5.3) has been read, EOF(0) is true.

Example 12.2 :

```
READY
RUN
00010  REM  THIS PROGRAM LISTS ITSELF ON THE CONSOLE,
00020  REM  ASSUMING IT IS LOCATED IN A FILE CALLED "PROG",
00030  REM  WITH SUFFIX "SA" ON DISK DRIVE 1.
00040  REM
00050  OPEN #10,"PROG:1",I \ OPEN ITSELF, SE QUENTIAL INPUT
00060  A$="" \ INITIALIZE VARIABLES
00070  B$=""
00080  C$=""
00090  INPUT #10 ,A$,B$,C$ \ READ ONE RECORD
00100  IF  EOF(10) THEN STOP "END OF FILE" \ EXIT WHEN EOF
00110  PRINT A$;B$;C$ \ NOT EOF, WRITE RECORD TO CONSOLE
00120  GO  TO 60 \ GO READ NEXT RECORD
00130  END

STOP END OF FILE


READY
```

Note that this example is the program listing AND the execution too.

Example 12.3 :

```
 10  REM  PROGRAM TO CONVERT A NUMBER IN ROMAN NUMERALS.
 20  REM
 30  DATA 1000,"M",900,"CM",500,"D",400,"CD"
 40  DATA 100,"C",90,"XC",50,"L",40,"XL"
 50  DATA 10,"X",9,"IX",5,"V",4,"IV"
 60  DATA 1,"I"
 70  REM
 80  INPUT "GIVE ME A NUMBER ( 0 TO STOP ) ",N
 90  REM
100  IF N<=0 THEN  STOP
110  RESTORE                         \ rewind data pointer
120  IF  EOF(0) THEN 180       \ if end of data , all done.
130  READ I,A$   \ fetch a test value and roman equivalent
140  IF I>N THEN 120 \ go fetch next test value if too big
150  N=N-I       \ update number by the current test value
160  PRINT A$;  \ print the roman equivalent of test value
170  GO  TO 140  \ go see if value can be subtracted again
180  PRINT                    \ end of roman numeral output
190  GO  TO 80                \ go prompt user on next line
200  END

READY
RUN
GIVE ME A NUMBER ( 0 TO STOP ) ?1979
MCMLXXIX
GIVE ME A NUMBER ( 0 TO STOP ) ?4602
MMMMLCII
GIVE ME A NUMBER ( 0 TO STOP ) ?0

STOP
```

12.5 Output transfer to file via the PRINT statement

To output data to a file, an extension of the PRINT statement is used whose general form is given below :

```
PRINT #LU AT key USING format, expl dell ... expn deln
```

See paragraph 5.5 for the description of expl to expn and dell to deln.
See chapter 6 for the "USING format," optional clause description.
See paragraph 12.3 for the description of LU parameter and for the "AT key" optional clause description.

The effect of this statement is the output of the variable list content to the file opened on logical unit number "LU", at the record numbered "key".
If the file is opened in input mode, a fatal error occurs.

If the positioning clause is not specified or ignored (see paragraph 12.3), the output transfer is made to the currently pointed record. If one record is not sufficient to hold all data output, the remaining data will be stored in the consecutive records (see paragraphs 5.6 and 12.2). Consecutive records in an indexed file are records whose keys are consecutive.

Remember that indexed or random files have fixed length records: If the record is not completely filled with the output data, trailing blanks are added up to the record length (These spaces may appear as data if read back in a string variable by an INPUT statement).

Writing to an unexisting record (or unknown key) extends the data file space (and the index table) automatically.

Care must be exercised when accessing sequentially a non-sequential file: output transfers may completely destroy the file structure and/or the index table.

Example 12.4 : A deck of punched cards has been accidentally shuffled! The shuffled card deck image has been put in a sequential disk file by an external program. Providing the cards are numbered from 10 to 20000 by step of 10 in columns 1 to 5 and there is a space in column 6, re-build the original card deck image in the same file.

```
 10  OPEN #3,"CARDS",U              \ update original file
 20  OPEN #25,"TEMP",U, IND ,80       \ indexed work file
 30  REWIND #3  \ update sequential file - read it first
 40  A$=""                          \ initialize variables
 50  B$=""
 60  C$=""
 70  INPUT #3 ,NUMBER,A$,B$,C$        \ input a card image
 80  IF  EOF(3) THEN 110 \ file transferred to work file
 90  PRINT #25 AT NUMBER A$;B$;C$    \ key is card number
100  GO  TO 40                      \ go input next record
110  REWIND #3                    \ rewrite sequential file
120  FOR NUMBER=10 TO 20000 STEP 10
130  INPUT #25 AT NUMBER ,A$,B$,C$        \ read work file
140  IF  EOF(25) THEN  STOP "CARD MISSING"
150  PRINT #3  USING "[C5] ",NUMBER;A$;B$; TRIM$(C$)
160  NEXT NUMBER                    \ transfer next record
170  STOP "DONE"                  \ card deck sorted, exit.
180  END
```

12.6 The REWIND statement

This statement is used to position the record pointer of a file at the first record in this file.

General form : REWIND #LU

See paragraph 12.3 for the description of the LU parameter.

The REWIND statement has no effect when applied to an indexed file.

REWIND #0 is equivalent to the RESTORE statement (See paragraph 5.3).

12.7 The CLOSE statement

This statement is used to close a file and release its assigned logical unit number. CLOSE can also be used to provide for file truncation and deletion.

General form : CLOSE #LU (normal)

CLOSE #LU,T (truncate)

CLOSE #LU,D (delete)

See paragraph 12.3 for the description of the LU parameter.

Although this is done implicitly by the STOP and END statements and by the normal termination process of a BASIC-M program, it is often needed to close a file before terminating a program, for example to open another file or to change the open mode. The CLOSE statement allows it.

The table below summarizes the actions performed on the file for the second and the third form of the close statement. The actions taken depend on the file open mode.

OPEN MODE	T	D
SEQ RAN Input IND	normal file closing	normal file closing
IND Output Update	normal file closing	file deleted
SEQ Output RAN	normal file closing	file deleted
RAN Update	file is truncated after the last referenced record	file deleted
SEQ Update	file is truncated after the last referenced record although the file pointer is positioned to the logical end of the file upon file opening	file deleted

NOTES:

- the "last referenced record" is the higher order record which has been read or written (not the one that chronologically preceded the CLOSE statement).
- CLOSE #LU,T without any prior reference to the file, deletes the file.

Example 12.5 : To concatenate several files and store in another one (MERGE source files).

```
 10  OPEN #3,"RESULT",O                    \ create result file
 20  DATA "FILE1","FILE2","FILE3","FILE4","FILE5"
 30  IF  EOF(0) THEN  STOP "DONE"      \ no more exist, exit
 40  READ A$                              \ fetch a file name
 50  OPEN #4,A$,I      \ open input mode, sequential access
 60  IF  EOF(4) THEN 140   \ file not found, open next file
 70  A$=""                              \ initialize variables
 80  B$=""
 90  C$=""
100  INPUT #4 ,A$,B$,C$    \ read a record from source file
110  IF  EOF(4) THEN 140 \ end of input file, go open next
120  PRINT #3 A$;B$;C$ \ output record to destination file
130  GO  TO 70                          \ go input next record
140  CLOSE #4                    \ close current source file
150  GO  TO 30                      \ go open next input file
160  END
```

12.8 Alphanumeric access key

The BASIC-M user will sometimes need to index files with alphanumeric strings: BASIC-M does not provide such a facility since for each application, the user may find a specific and appropriate coding algorithm. However, the following example illustrates the use of an indexed file as a phone directory. The first program must be used to enter or modify records in the file. The second program is executed to consult the phone directory. A hashcoding routine is called before each data transfer with the file to find the numeric access key assigned with the given string.

Example 12.6 : Phone directory

```
 10  INTEGER AKEY,CHAR            \ Hashcode routine variables
 20  GO  TO 200                                \ skip procedure
 30  DEF HASH(KEY$,LU)
 40  AKEY=0                              \ initialize access key
 50  A$=KEY$+"                                    " \ 31 chars.
 60  FOR I=1 TO 31                         \ use all characters
 70  CHAR= ASC( MID$(A$,I,1))            \ next character code
 80  AKEY= IOR( ISHFT(AKEY,3), ISHFT(AKEY,-13))    \ rotate
 90  AKEY= IEOR(AKEY,CHAR) \                3 bit left, 13 right
100  NEXT I                \ same with next character code
110  INPUT #LU AT AKEY ,B$                          \ read record
120  IF  EOF(LU) OR (A$=B$) THEN  RETURN
130  AKEY=AKEY+1    \ redundant definition, see next record
140  GO  TO 110          \ loop until found or empty record
150  REM
160  REM --MAIN PROGRAM--
170  REM
200  OPEN #4,"PHONE",U,IND,42  \ open phone directory file
210  INPUT "GIVE NAME AND PHONE NUMBER ",NAME$,PHONE
220  IF NAME$="END" THEN  STOP               \ type END to quit
230  HASH(NAME$,4)             \ compute numeric access key
240  PRINT #4 AT AKEY  USING "[31][X][9]",NAME$,PHONE
250  GO  TO 210                  \ recorded, go input next name
```

```
 10  INTEGER AKEY,CHAR           \ Hashcode routine variables
 20  GO  TO 200                                \ skip procedure
 30  DEF HASH(KEY$,LU)          \ same as in creation program
 40  AKEY=0
 50  A$=KEY$+"                              "
 60  FOR I=1 TO 31
 70  CHAR= ASC( MID$(A$,I,1))
 80  AKEY= IOR( ISHFT(AKEY,3), ISHFT(AKEY,-13))
 90  AKEY= IEOR(AKEY,CHAR)
100  NEXT I
110  INPUT #LU AT AKEY ,B$
120  IF  EOF(LU) OR (A$=B$) THEN  RETURN
130  AKEY=AKEY+1
140  GO  TO 110
150  REM
160  REM --MAIN PROGRAM--
170  REM
200  OPEN #4,"PHONE",I,IND      \ open phone directory file
210  INPUT "WHO DO YOU WANT TO PHONE TO ",NAME$
220  IF NAME$="END" THEN  STOP              \ type END to quit
230  HASH(NAME$,4)             \ compute numeric access key
240  IF  EOF(4) THEN 300                   \ test empty record
250  INPUT #4 AT AKEY ,NAME$,PHONE      \ read phone number
260  PRINT  TRIM$(NAME$);"'S PHONE NUMBER IS ";PHONE
270  GO  TO 210                  \ go ask for another name
300  PRINT  TRIM$(NAME$);" IS NOT IN THE PHONE DIRECTORY"
310  GO  TO 210                  \ go ask for another name
```

12.9 Array input/output with disk files

The MAT INPUT (paragraph 9.3) and MAT PRINT (paragraph 9.4) statements may be applied to disk files too. This is discussed in this paragraph.

12.9.1 Input of an array from a file

General form : MAT INPUT #LU, Arr

where :

LU	is the logical unit (See description in paragraph 12.3).
Arr	is the name of the array in which the input data will be stored.

This statement is equivalent to the BASIC-M-like sequence :

```
FOR Cntr = 1 TO Number_of_rows_in_Arr
INPUT #LU,Arr(Cntr,1),...,Arr(Cntr,Number_of_columns_in_Arr)
NEXT Cntr
```

One can notice that positioning the pointer in a file is not possible within this statement. For this reason, arrays may not be properly input from an indexed file, since the absence of index key is interpreted as a zero value key. This means that each row of the array will be read from the zero key record.

12.9.2 Output of an array to a disk file

General form : MAT PRINT #LU Arr

where :

LU	is the logical unit (See description in paragraph 12.3).
Arr	is the name of the array to be stored in file.

See paragraph 9.4 for the exact definition of operations.

As for the MAT INPUT statement, the index key cannot be specified; for this reason, a MAT PRINT statement execution on an indexed file will store the first array row in the record of zero value key, and the other rows in the records consecutively numbered.

Example 12.7 : This is a complete program to test the validity of a specific matrix inversion. The user may choose the output device at execution time without modifying the program. The output device may be the console, the line-printer or a disk file. In the latter case, the user is prompted for the output file name.

```
 10  DATA "CN",1,"cn",1,"LP",2,"lp",2,"FILE",99,"file",99
 20  INPUT "GIVE OUTPUT DEVICE (LP, CN OR FILE) ",DEVICE$
 30  REM  Strip leading and trailing blanks
 40  IF  SUBSTR(DEVICE$," ")<>1 THEN 70
 50  DEVICE$= TRIM$( RIGHT$(DEVICE$, LEN(DEVICE$)-1))
 60  GO  TO 40
 70  IF  EOF(0) THEN 110               \ see if legal input
 80  READ NAME$,LU
 90  IF NAME$=DEVICE$ THEN 140         \ if found, exit loop
100  GO  TO 70
110  PRINT "ANSWER CORRECTLY, PLEASE"    \ bad input, reask
120  RESTORE
130  GO  TO 20
140  IF LU#99 THEN 170         \ if disk, ask for a file name
150  INPUT "GIVE FILE NAME PLEASE ",FILE$
160  OPEN #LU,FILE$,O, SEQ             \ create output file
170  PRINT #LU  USING 190,"MATRIX INVERSION"     \ device ok
180  PRINT #LU  USING 190,"----------------" \ print header
190  IMAGE "[78,C]"
200  PRINT
210  INPUT "MATRIX DIMENSION ",N
```

```
220  IF N>0 AND N<11 THEN 250
230  PRINT "MATRIX DIMENSION ALLOWED BETWEEN 1 AND 10"
240  GO  TO 210 \ REASK FOR CORRECT INPUT
250  MAT A= ZER(N,N)            \ initialize matrix dimensions
260  PRINT #LU "  MATRIX DIMENSION : ";N;"X ";N  \ echo DIM
270  PRINT #LU
280  INPUT "GIVE INPUT DATA FILE NAME",DATA$
290  OPEN #100,DATA$,I, SEQ                  \ open input file
300  IF  NOT( EOF(100)) THEN 340             \ file exists, ok
310  PRINT "FILE NOT FOUND"
320  CLOSE #100
330  GO  TO 280                         \ reask for file name
340  MAT  INPUT #100 ,A                  \ input matrix data
350  PRINT #LU "INPUT MATRIX"
360  MAT  PRINT #LU A           \ echo data on output device
370  MAT B= INV(A)                           \ invert matrix
380  PRINT #LU "MATRIX INVERSE"
390  MAT  PRINT #LU B       \ print inverse to output device
400  MAT C=A*B        \ multiply: must find identity matrix
410  PRINT #LU "    A*B"
420  GOSUB 470
430  MAT C=B*A        \ identity must be found this way too
440  PRINT #LU "    B*A"
450  GOSUB 470
460  STOP "DONE"
470  MAT  PRINT #LU C         \ print multiplication result
480  RMS=0   \ compute accuracy of the previous calculation
490  MIN=1.E38
500  MAX=0
510  FOR I=1 TO N                         \ scan full matrix
520  FOR J=1 TO N
530  E=0                          \ compare to identity matrix
540  IF I=J THEN E=1
550  E= ABS(C(I,J)-E)
560  RMS=RMS+ SQ(E)
570  IF MIN>E THEN MIN=E
580  IF MAX<E THEN MAX=E
590  NEXT J
600  NEXT I
610  RMS= SQR(RMS/ SQ(N))
620  PRINT #LU  USING 630,"STATISTICS",MIN
630  IMAGE "[23,C][/2]BEST  : [2,8,3]"
640  PRINT #LU  USING 650,MAX,RMS
650  IMAGE "WORST : [2,8,3][/]RMS   : [2,8,3]"
660  PRINT #LU
670  RETURN
680  END
```

CHAPTER 13

13. SYSTEM COMMANDS

This chapter covers the commands used to invoke BASIC-M, create a source program, run it and save it and compile it. There are several system parameters, such as the compiler re-entry point, which are dependent on the implementation; they are hereafter referred to by symbols, rather than their absolute address. The user is requested to carefully read the instructions which are given separately when purchasing BASIC-M, in order to know the absolute values of the system parameters. These are available on a "NEWS" file in the system disk.

13.1 Operating Modes

There are two possible modes of operation with BASIC-M.
The first mode is Interpreter Mode, in which new source programs can be created and immediately executed using the RUN command. The interpreter mode can also be used to load an old source program, possibly created by the CRT editor, from disk, and to execute this program with the RUN statement. The prime advantage of Interpreter mode is that it allows the user modify the source and subsequently save this to disk either overwriting the old file or creating a new one. This permits a fast change-tryout-change iteration when writing new software.
Programs to be executed in Interpreter mode are restricted in use of many additional features of BASIC-M such as Interrupt Handling, use of assembly sub-routines, real external addresses, as they may interfere with BASIC-M´s operation. Nor can they access the graphic RAM area of the EXORset.
For these programs, the source can be created either using the CRT editor or within BASIC-M, and partially debugged in Interpreter Mode. Then the programs should be compiled by invoking BASIC-M in Compiler Mode (option O), in order to create an object module on disk. The size of these programs can be larger than is possible in Interpreter Mode,as they need not be entirely resident in memory at compile time.

13.2 Invoking BASIC-M

The BASIC-M compiler/interpreter is invoked under control of the disk operating system; once this latter has been loaded, the "=" prompt sign is displayed. The operator can then invoke BASIC-M by typing the following command:

```
=BASICM <name 1>[,<name 2>][;<options>]
```

where :

<name 1> is the name of the file to be further executed, possibly after some BASIC-M editing,
<name 2> optionally specifies the name of an output file which is created when BASIC-M is exit.
Both file specifications are in the standard disk-operating system format:

<file name> [.<suffix>] [:<logical unit number>]

The default values "SA" and zero are used for the suffix and the logical unit number, respectively, if they are not explicitly entered.

The following options are valid:

If no options are specified BASIC-M operates in Interpreter Mode.

!	Autostart, i.e. chain the loading, compilation and execution of an existing source file whose name is <namel>. This option is mutually exclusive with all the other options defined below. After execution of the user program, control is returned to the operating system.
O[=:DRV]	Invoke BASIC-M in the compiler mode. file <namel> must exist. file <name2> must not be specified. In the compiler mode, BASIC-M returns the object code in a file whose name is <namel>, whose suffix is ".LO", and which is constructed in the same drive as the source file <namel>, unless a destination drive number DRV is specified. In this case, the object file and the compiler scratch file are both constructed on this drive.
-O	Invoke BASIC-M in the compiler mode without generation of a user object code file; "-O" is used to get a compilation listing only.
S	Produce a compacted object code (see paragraph 13.3.13).
M	Output a symbol table to the listing device.
-M	Do not output the symbol table.
L[=#CN]	Output compilation listing to console.
L=#LP	Output compilation listing to printer.
L=<name3>	Output compilation listing to the file whose name is <name3>, whose suffix is ".BL". Destination drive defaults to drive 0.
-L	Do not output a compilation listing. Message and error indications, if any, will be displayed on the console.
R=$XXXX	Produce object code to execute in conjunction with

the Runtime package based at address XXXX in the end user system.

D=$YYYY User Data Section base address.

P=$WWWW User Program Section Origin. WWWW is the load and start address of the user object file; this information is saved in the object file RIB.

Default options for compiler mode:

-S, M, -L, L=#CN if L only specified
R=$6B00
D=$2000
P=$4000

Each record of the input file must be numbered (line numbers are in the range 1 to 65535), and should not contain more than 80 ASCII characters.

If the diskette file <name 1> already exists, the input will be taken from it. If <name 1> does not already exist, then it will be automatically created, and the user program may be further saved to it upon exit.

The second file name specification can only be used if the file <name 1> to be edited already exists on the diskette. The output file is used to receive the user program <name 1> after it has been edited and/or run by BASIC-M. When BASIC-M is exit, the output file contains a complete copy of the input file plus any changes that were made to the BASIC-M source program once it was loaded in the workspace buffer. The input file is preserved.

One of the standard operating system error messages will be displayed if the input file <name 1> is delete or write protected and <name 2> is not specified, or if the output file <name 2> already exists.

The following are examples of BASIC-M valid invocations:

```
=BASICM DEMO:1 (1)
=BASICM:1 TEST.BS, NEWTEST.BM (2)
```

A slightly different form makes provision for chaining automatically the loading, compilation and execution of an existing BASIC-M source program. For instance, to execute a program named "DEMO" residing in drive 1, (1) could be typed in as :

```
=BASICM DEMO:1;!
```

Likewise, (2) would be :

```
=BASICM:1 TEST.BS;!
```

13.3 Interpreter Mode

13.3.1 Creating the source program

The BASIC-M source program can be edited either "off-line" under control of the system text editor, or "on-line" under control of the BASIC-M editor. For editing long programs, it is recommended to use the system text editor which provides more facilities than the BASIC-M editor; the user must keep in mind that his source program must be line-numbered. For a complete description of the system editor, refer to the relevant manual.

The BASIC-M editor is line-oriented. It provides for line insertion, deletion or replacement. Again, it is emphasized that all the statement lines must start with a valid line number (in the range 1 to 65535), followed by at least one space character.

If the user wishes to insert a statement between two others, he types a statement number that falls between the other two followed by the statement he wishes to insert. After the statement has been completely entered, the user enters a carriage return to complete the insert.

If the user wishes to delete a statement line, he merely enters the number of the statement followed by a carriage return.

If the user wishes to replace a statement, the number of the statement to be replaced must be typed, followed by the new statement and a carriage return.

When a statement is being typed in, the user may delete the last entered character by hitting the "RUBOUT" (the character just deleted is echoed to the console). The whole statement line may be cancelled by striking the "X" key while holding down the "CTRL" key.

13.3.2 Auto line-numbering

The "N" command requests BASIC-M to automatically output line numbers.

N [N1] [,N2]

Where :

-"N1" is the first line number to be prompted.
-"N2" is the value to be added to "N1" to form each succeeding line number prompted.

The "N" command initiates the input process in which all data following the command is inserted into the BASIC-M workspace buffer. The two parameters N1 and N2 default to the value 10. Prompting will continue until a carriage return is entered as first character of a statement.

13.3.3 RESEQuence

This command is used to renumber the statement lines of a program.

Syntax : RESEQ [N1]

where :

"N1" is the line number at which to begin resequencing and the increment to be applied to form each subsequent line number within the source program.

N1 defaults to the value 10.
Prior to actually resequencing, BASIC-M checks that the highest line number resulting from the resequence operation does not exceed the allowable range (65535); if this condition is not met, the message "UNABLE" is displayed to the console.

13.3.4 LIST and LIST Erroneous statement lines

The "LIST" command allows the display of all or a portion of the source program.

Syntax : LIST [#LU] [N1-N2]

where :

-N1 and N2 specify the first and last statement lines respectively, to be listed.
-LU specifies the output device.

The list defaults to the entire program to be displayed to the console (LU=1); LU should be set equal to 2 in order to direct the printout to the system line printer.

The "CTRL-W" and "CTRL-P" codes can be entered while the program is being listed to suspend, or respectively abort, the list operation. Once the printout has been temporarily suspended with the "CTRL-W" code, it can be resumed by striking any key.

Statements whose syntax is incorrect appear as REM statements, with the error code listed after the REM keyword. The following is an example of the LIST command:

```
READY
```

```
LIST 25-55

00027  REM ... INPUT STRING ...
00035  INPUT A$
00043 REM **13** 00043 PRINT A$+B+CHR$(7)
00051  GOTO 35

READY
```

The above listing shows that the statement line 43 has an error whose code is 13. The user still does know where the error resides within the line. To detect the level of the error, he may ? then use the LISTE command which is syntaxically the same as the simple LIST command.

LISTE displays only those statements which have been flagged as syntactically erroneous. A pointer points to the token in error. The following listing would be obtained on the system line printer if the LISTE command was applied to the previous sample program:

```
READY
LISTE #2 25-55
................................V
00043 REM **13** 00043 PRINT A$+B+CHR$(7)

00001 ERRORS       ( B is a numeric variable, and
                     therefore cannot be embedded
READY                in a literal expression. )
```

13.3.5 FLAGON and FLAGOFF

Normally, syntax errors, if any, are not reported until the program is listed with the LIST or LISTE errors. A command exists for the user to be informed immediately of syntax errors, as each statement line is entered. This immediate detection is activated by giving the command "FLAGON", which can be further disabled by entering its counterpart "FLAGOFF" command (default state). An example is presented next:

```
READY
FLAGON
N 27,8
   27 REM ... INPUT STRING ...
   35 INPUT A$
   43 PRINT A$+B+CHR$(7)
................^
ERROR  #13
   51
```

13.3.6 The DELete command

This command permits the user to delete a block of lines in the program using a single command. A single line can be deleted by entering the line number immediately followed by a carriage return. To delete a block of lines the syntax is as follows:

```
DEL N1 [-N2]
```

Where: N1 is the line number of the first statement to be removed
N2 is the line number of the last statement to be removed.

13.3.7 The RENAME command

As is implied, this command is used to rename the variables of the source program, if the user so desires. This is a convenience for changing the one- or two-character variable names of a standard BASIC program into more meaningful names.

Syntax : RENAME VAR1 VAR2

where :

VAR1 stands for an existing variable whose name is to be changed into VAR2.

VAR1 and VAR2 must be of the same type (numeric or string).
VAR2 must not have been used previously in the program.
The substitution is not applied to the names which may be defined in comment lines or in literal constants.

Below is an illustration of this command :

```
LIST

00010   REM ... INPUT STRING A$ ...
00020   INPUT A$
00030   PRINT A$+B$
00040   GOTO 20

READY
RENAME A$ NEW_NAME$

READY
LIST
00010   REM ... INPUT STRING A$ ...
00020   INPUT NEW_NAME$
00030   PRINT NEW_NAME$+B$
00040   GOTO 20
```

13.3.8 Returning to the disk-operating system

Once a session is terminated, the user may return to the disk-operating system by using the "QUIT" command. When this command is entered, the user is requested whether he wishes to save his source program (to the output file defined by the command line which was entered to invoke BASIC-M) (see 13.1). If the answer is "Y", the program is first dumped to the output file, and the disk-operating system is then re-entered (the prompt "=" is displayed). If the answer is "N", step 2 only occurs. If the answer is not satisfactory, the question "SAVE(Y/N) ?" is issued again.

IMPORTANT NOTE

BECAUSE PROGRAMMING ERRORS MAY CAUSE THE ALTERATION OF THE MEMORY-RESIDENT SOURCE PROGRAM AT EXECUTION TIME, THE USER IS HIGHLY ENCOURAGED TO SAVE HIS SOURCE ONTO A DISKETTE FILE PRIOR TO ISSUING THE "RUN" COMMAND.

13.3.9 The RUN command

The execution of a BASIC-M source program is invoked by the "RUN" command, which operates internally in three steps:

1/ If the source program has already been compiled successfully, control is transferred directly to step 3 below.

2/ Otherwise, the source program is first compiled thus producing a position-independent object code.

3/ The object code is executed under control of the run-time package. Execution proceeds until one of the following conditions is met:

- -the last statement line has been executed.
- -a STOP or END statement is encountered.
- -a fatal error occurs.
- -the operator aborts the execution by entering the "CTRL-P" code.

Either of the above conditions will cause the execution of the object code to terminate and control to be transferred back to BASIC-M. Note also that typing "CTRL-W" causes the execution to be suspended until another keystroke causes it to resume.

If the RUN command is entered again without the source program being modified meanwhile, step 1 of the RUN process depicted above is omitted, and the overall execution will proceed slightly faster.

13.3.10 TRON and TROFF

The TRON command is used to trace a BASIC-M program, with each statement line number being displayed prior to its execution. The TROFF command cancels a TRON request.

13.3.11 The PATCH command

The PATCH command is used to exit temporarily from BASIC-M and transfer control to the system monitor. Once in the monitor, BASIC-M may be re-entered by typing the proceed command ";P".

13.3.12 The NEW command

The NEW command causes the working storage area in memory and pointers to be reset. The effect of using this command is to erase all traces of the program currently stored in memory in order to start over.

13.3.13 The COMPILE command

The COMPILE command allows the user to generate a compiler listing when operating in interpreter mode.

Syntax: COMPILE [<options,>]

Where the possible options are:

"S" request code generation optimization
"M" display symbol table
"L" print the compile address of each line
"R" specify the base address of the runtime package.
"D" specify the base address of the data section.

the effect of these options is discussed next

Option "S" : Normally, each executable statement line compiles into the following:

```
JSR RUN1
FDB statement line number
   :
code reflecting
the statement
   :
```

RUN1 is a subroutine in the runtime package which, in particular, takes care of displaying statement numbers when the trace mode is active, of checking for stack overflow, operator abort or suspend, and of testing conditions associated with the WHEN ... THEN statements.
.The option "S" prevents the compiler from generating the first two lines shown in the above code expansion, thus providing for a saving of 5 bytes per statement line ... and for faster execution. Again, it must be stressed that this option suppresses the

following features during program execution:

-line number printing on error.
-WHEN...THEN monitoring.
-stack checks.
-operator´s action checks.
-trace.
Therefore, this option should be used essentially to recompile programs which have proven error free, and which do not contain WHEN statements.

Option "M" : Causes the printout of the symbol table which shows the attribute and location of each variable / procedure / function defined within the BASIC-M program.
The following attributes are defined:

-B : byte variable.
-I : integer variable.
-R : real variable.
-S : string variable.
-RF: real user-defined function.
-SF: string user-defined function.
-P : user-defined procedure.
-E : external function/procedure.

In addition, this option also causes the printout of the data section (RAM) and program section (ROM) limits of the compiled program.

Example :
```
10  BYTE Screen(22,80) ADDR $E000
20  INPUT A$
30  DEF SUM(X,Y)=X+Y
40  PRINT USING A$, SUM(3,8)

COMPILE S,M

NO ERROR                             dimension

                                         v
Screen ...............B......E000....2
A$ ...................S......990C.....
SUM ..................RF.....002C.....
X ....................R......992C.....
Y ....................R......9931.....

DSCT: 990A-9D81
PSCT: 9D82-9E1B
```

Option "L" : To print the absolute compile address of each statement, so that breakpoints can be inserted at the beginning of each line of the program.

Example : compiling the previous program.

```
COMPILE S,L

00010....9D97
00020....9D97
00030....9DAE
00040....9DD2
```

Option "R" : To specify the absolute base address of the runtime package in the end-user system (the package is position-independent).

Option syntax : R=$nnnn , where

$nnnn is the hexadecimal base address of the runtime package.

NOTE : PROGRAMS COMPILED WITH THIS OPTION MUST NOT BE EXECUTED WITH THE "RUN" COMMAND.

Option "D" : To specify the absolute base address of the scratchpad RAM (the one storing the BASIC-M variables and stacks) in the end-user system.

Option syntax : D=$mmmm , where

$mmmm is the hexadecimal base address of the scratchpad RAM.

NOTE : THE FIRST 34 BYTES OF MEMORY (LOC 0 TO $21) ARE RESERVED.

The following system dependencies are defined. Refer to the attached documents for reading their absolute addresses :

RUN.ST : Run-time package start address.
RUN.EN : Run-time package end address.
BASICM : BASICM warm-start entry point.

13.4 Compiler mode

Most of the time, BASICM is used in the interpreter mode because this mode provides the desirable interaction for the operator to quickly write and debug small to medium-sized programs. In this mode, the system memory is shared as follows :

```
ENDUS     _______________________
         |     symbol table      |
         |-----------------------|--|
         |                       |  v
         |-----------------------|
         |  object code   PSCT   |
         |-----------------------|
         |  object data   DSCT   |  ^
         |-----------------------|--|
         |        I-code         |
RUN.ND   |-----------------------|
         |                       |
         |       Runtime         |
OVERW    | - - - - - - - - - - - |
RUN.ST   |-----------------------|
         |                       |
         |       Compiler        |
         |                       |
         |-----------------------|
         |   compiler buffers    |
$2000    |-----------------------|
         |                       |
         |   Operating system    |
 0000    |_______________________|
```

where :

ENDUS is the address of the last location of contiguous RAM
The symbol table expands towards 0000.
I-code is the buffer storing the intermediate source code, that expands towards the symbol table.
RUN.ST and RUN.EN denote the starting and ending address, respectively, of the Runtime package as loaded by the BASICM system command.

As is shown, BASICM allocates storage for object data and object code in the area which lies in between the upper end of the I-code buffer and the top of the symbol table. Clearly, as the source program expands and/or as the object data section requires more and more memory space, one may end up in a situation where the memory space left cannot accommodate the object code section. This situation results in the message "NO ROOM" being displayed when attempting to "RUN" the source program. It is then suggested to use BASICM in the compiler mode. In this mode of operations, BASICM will allocate the buffers mentioned above from the location labelled OVERW in the map shown before. This extends the available space for the compiler by 12K bytes approximately. Since most of the Runtime package is overwritten in this mode, the execution of the object program cannot be directly envoked. The user will have to merge the produced object code and the runtime package prior to loading and executing the resulting module. Note that the environment parameters (DSCT address and Runtime origin) must be specified when compiling the source program.

The Compiler Mode is activated by envoking BASIC-M with the following command line :

```
=BASICM <name>;O[=:DRV][S][M][L][R=$xxxx][,D=$xxxx][,P=$xxxx]
```

where :

<name> is the name of the input file to be compiled, and conforms to the file specifications set in section 13.1.
"O" (standing for object output) is the option activating the compiler mode. Refer to section 13.2 for a description of the other options.

The following is an example:

```
=BASICM SAMPLE:1;OSMLR=$8400,D=$100,P=$3000

BASIC-M INTERACTIVE COMPILER
 COPYRIGHT BY MOTOROLA 1979
    EXORset release 3.00


---
PAGE 01   SAMPLE   .SA:1

00010  INPUT A,B
00020  PRINT A+B
00030  IF A=0 THEN STOP
00040  GOTO 10


A ...................R..........0100.....
B ...................R..........0105.....

DSCT: 0100-0555       PSCT: 3000-3085
RUNTIME BASE : 8400
END OF COMPILATION
```

The user program SAMPLE.LO will execute when loaded with the runtime package of BASIC-M. The BLOAD utility can be used to load the user module together with the runtime package, or to merge them into one file that can be later downloaded into a target system.
Note that for programs to execute in the EXORset under XDOS, the data section, program section and runtime package must be located above $2000. Otherwise they will overwrite XDOS.

13.5 The BLOAD Utility

The BLOAD utility is available with BASIC-M as a .CM file on the the system diskette. Its function is to merge a user module or modules with the runtime package and either create a disk file or load the module into memory for immediate execution.
BASIC-M must be resident in drive 0 during the operation of BLOAD.

Syntax: BLOAD <name 1>[,<name 2>, ... ,<name n>][;<options>]

Where: <name 1> is the name of a user code file generated by the BASIC-M Compiler.

<name 2>, ...,<name n> are the names of object files to be loaded/merged with <name 1> and the Runtime package. These can be assembly-language written or BASIC-M written routines.

<options> are as follows:

Option	Default	Function
O[=<concat>]	-O	Merge all files <name j> (j=1 to n) and the Runtime package in to a single object file <concat>. <concat> defaults to file <name 1> with "CM" as suffix.
G	-G	Load all files <name j> (j=1 to n) and the Runtime package, and execute program from origin of <name 1>.
L=$XXXX	L=$0020	Patch the first 2 bytes of the Runtime package with the value XXXX which represents the address of DCST LINK where the Runtime memorizes the origin of the user data section.
R		Extract the Runtime package from the BASIC-M file on drive 0 and load it in memory at the address implied by the user object file <name 1>. "R" implied if option "G" is selected. If option "O" is selected and "G" is not the Runtime is not loaded/merged.
M	M	Include matrix operations in Runtime.
D	D	Include disk operation in Runtime. The combination "-MD" is not allowed.

Notes:

If niether option "O" nor option "G" are selected, the object files and the Runtime package are loaded into memory and control is passed to the debug monitor.

Object files may be loaded over the disk operating systyem and/or the BLOAD command provided that option "O" is selected.

As the Runtime package is extracted from the BASIC-M compiler during the load / merge process, BLOAD expects to find the file "BASICM.CM" on drive 0. As a result, make sure that this file is available on drive , and furthermore never rename BASIC-M.

Example:

The following is an example of creating a disk command (suffix "CM").

Example problem: The STOP statement causes the system monitor to be reentered if the program is run on the EXORset, while control is transferred to MDOS if the p rogram is run on the EXORciser. If the Exorset user desires to return to the XDOS operating system, he can program a procedure to do this. A possible solution is shown below:

```
=BASICM DOS:1;OLMS

BASIC-M INTERACTIVE COMPILER
 COPYRIGHT BY MOTOROLA 1979
   EXORset release 3.00




---

PAGE 01  DOS     .SA:1

0011   00010   INTEGER DOSVEC
0011   00020   EXTERNAL DOS ADDRESS DOSVEC
0011   00030   DOSVEC=$3F1A            \"SCALL  .MDENT"
0024   00040   REM
0024   00050   INPUT A,B
003D   00060   PRINT A,B
0060   00070   IF A=0 THEN DOS
0083   00080   GOTO 50
0088   00090   END

DOSVEC......................I......2000.....
DOS.........................E......2000.....
A...........................R......2002.....
B...........................R......2007

DSCT: 2000-24BB              PSCT: 4000-40A2
RUNTIME BASE : 6B00
END OF COMPILATION

=BLOAD DOS:1;O=DOS:1
DOS     .LO:1  4000 - 40A7
RUN-TIME       6B00 - A2F7
DATA SECTION   2000 - 24BB

LOAD BOUNDS    4000 - A2F7
=DOS:1
? 1,1
1                 1
?0,1
0                 1
=
```

Rentering the operating system as shown above does not cause the open files to be closed; therefore, if files have been opened by the BASIC-M program, the user should provide the necessary statements to get them closed.

NOTE: BASIC-M programs with system calls must not use the first 8K bytes of memory.

BASIC-M programs with disk I/O imply that the operating system be loaded in memory

CHAPTER 14

14. PERFORMANCE CHARACTERISTICS

14.1 Requirements

The disk-version of BASIC-M runs on the EXORciser / EXORterm, or EXORset development tools equipped with a minimum of 48 kilobytes of RAM. The supported disk-operating system (MDOS or XDOS) is used for loading and saving the source programs, and for exchanging data with the diskettes when the BASIC-M programs contain disk input/output statements.

The Compiler / Interpreter and the runtime package occupy about 14K bytes of RAM each. The runtime package is ROM-able and position-independent.

14.2 Space estimates

As the ASCII source program is entered, BASIC-M takes each incoming line and processes it to an intermediate code which takes less memory than the original program and allows for a faster compilation. In this intermediate code, in particular, variable or function/procedure names are coded as pointers to a symbol table, line numbers and hexadecimal constants as 16-bit words, and each keyword as an 8-bit code. Hence, the following hints:

-feel free to use readable names.
-keep the number of comments and their length to a minimum (comments are reproduced "as-is" in the intermediate code).
-whenever possible, use hexadecimal constants.
-if a constant appears several times in the program, equate it to a variable and use this variable name to reference it.

When the RUN or BASICM;O commands are entered, the intermediate code is translated into the final object code, and memory is allocated to the variables. In order to minimize the size of the program (object code) and data sections (scratchpad), the following simple rules should be observed:

-do not omit to dimension arrays prior to referencing their elements with subscripted variable names.
-do not use real variables where byte or integer variables could be used.
-avoid mixed-mode expressions.
-compile the source program with the "S" option, whenever possible (refer to paragraph 13.11).

The compiled code uses approximately 1/3 to 1/2 as many bytes as the source text; this value is an estimate only and may vary in either direction from program to program.

14.3 Speed estimates

Herebelow are some execution times of a few runtime routines:

```
floating add :         950 us max.
floating multiply :      2.5 ms max.
floating divide :        6.7 ms max.
FIX and FLOAT :        ~ 270 us
trig functions :     ~ 20-30 ms
```

The following sample programs and results give more significant figures as far as speed is concerned.

Benchmark BK1

```
10   FOR K=1 TO 10
20   NEXT K
```

a/ Program compiled without option "S"
1.8 ms / loop

b/ Program compiled with option "S"
1.67 ms / loop

c/ Program modified to declare K as an integer, and to code the constant 10 as an hex constant ($A).
0.16 ms / loop

Benchmark BK2

```
10   K=0
20   K=K+1
30   IF K<10 THEN 20
40   STOP
```

a/ same as a/ in BK1
2 ms / pass

b/ same as b/ in BK1
1.73 ms / pass

c/ same as c/ in BK1
0.4 ms / pass

Benchmark BK3

```
10   K=0
20   K=K+1
30   A=K/K*K+K-K
40   IF K<10 THEN 20
```

a/ Program compiled without option "S"
    6.8 ms / pass
b/ Program modified to declare K and A as integer variables, and to code 10 as an hex constant. Option "S" is used.
    1.8 ms / pass

Benchmark BK4

```
300  PRINT "START"
400  K=0
430  DIM M(5)
500  K=K+1
510  A=K/2*3+4-5
520  GOSUB 820
530  FOR L=1 TO 5
535  M(L)=A
540  NEXT L
600  IF K<1000 THEN 500
700  PRINT "END"
800  END
820  RETURN
```

a/ Program compiled without option "S"
    20.8 sec
b/ Program compiled with option "S"
    18.9 sec

APPENDIX

A. ASCII Character Set

BITS 4 TO 6 --		0	1	2	3	4	5	6	7
	0	NUL	DLE	SP	0	@	P	`	p
	1	SOH	DC1	!	1	A	Q	a	q
B	2	STX	DC2	"	2	B	R	b	r
I	3	ETX	DC3	#	3	C	S	c	s
T	4	EOT	DC4	$	4	D	T	d	t
S	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
0	7	BEL	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
T	9	HT	EM	)	9	I	Y	i	y
O	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
3	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

APPENDIX

B. Syntax Error Messages

1	Invalid logical expression in an IF or WHEN statement
2	Missing THEN in an IF or WHEN statement
3	THEN must be followed by an executable statement
4	Uncomplete bit selector (missing "]")
5	Illegal procedure name or bit selector not followed by "="
6	Equal sign expected
7	Illegal branch statement
8	GOTO or GOSUB not followed by a valid line number
9	CALL is not followed by a valid procedure name
10	Missing ")" in an argument list, selector or array size
11	Illegal arithmetic expression
12	Missing ")" in an arithmetic expression
13	Illegal literal expression
14	Missing or invalid argument list in a POKE statement
15	Invalid unsigned integer constant
16	Invalid exponent
17	Filename must be a string variable or constant
18	Invalid variable name in a DIM statement
19	Illegal or unspecified array size in a DIM statement
20	Illegal ADDRESS clause in a DIM or EXTERNAL statement
21	Illegal variable name in a BYTE or INTEGER statement (string variable names not allowed)
22	Missing address specification in an EXTERNAL statement
23	Illegal operands in a READ statement No separator, or expression or illegal variable name

24	Illegal operands in a DATA statement Operand is neither numeric, nor hexadecimal, nor string
25	Missing "#" in an OPEN, CLOSE or REWIND statement
26	Missing comma in an OPEN statement
27	Undefined data transfer mode in an OPEN statement Neither I, nor O, nor U.
28	Illegal file access Neither SEQ, nor IND, nor RAN
29	"=" required in a LINE or DIGITS statement
30	Illegal index name in a FOR or NEXT statement
31	Index not followed by "=" in a FOR statement
32	Missing TO in a FOR statement
33	Invalid NEVER or ON statement
34	Invalid line numbers list in an ON ..GOTO statement
35	IMAGE not followed by a format string
36	Illegal variable or procedure name
37	Missing parentheses in a logical expression
38	Invalid file number in an EOF function
39	Invalid relational operator
40	Illegal or missing separator in an INPUT statement
41	Invalid key
42	No argument list following the TAB keyword, or Expression not enclosed between parentheses in a matrix scalar operation
43	Invalid operand in a MAT READ, INPUT or PRINT statement Expressions are not allowed
44	Missing comma in a MAT INPUT or MAT PRINT statement
45	Missing "=" in a matrix assignment statement
46	expression not enclosed in [ ] in a MAT SET statement
47	Missing argument in a MAT INV or MAT TRN statement
48	Illegal character scanned
49	Illegal statement
50	Statement too long

51 Missing argument list following a string function

....

APPENDIX

C. Compilation Error Messages

1	variable is redefined.
2	forward reference
3	first dimension is null or overflows
4	two dimensions specified while first one exceeds 255
5	second dimension is null or overflows
6	second dimension exceeds 255
7	more than 64 K are spanned
10	function redefined
11	DATA statement operand is not a constant
12	signed hexadecimal constant
13	constant overflows
14	exponentiation requires that one of the operands be real
15	invalid dimensioned variable or undefined user-function or procedure
16	bit reference does not apply to a BYTE or INTEGER variable
17	attempt to invert or transpose a simple variable
18	illegal call of a user-defined procedure / function
19	implicit redimensioning of a variable
20	variable not recognized during previous pass
21	user function is defined forwards
22	expression used as argument in the "LOC" function
23	too many arguments in a built-in function
24	built-in function does not support argument
25	missing argument in a built-in function

26	TAB is used in a "PRINT USING" statement
27	Too many nested "FOR-NEXT" loops (21 max.)
28	Nested "FOR-NEXT" loops with same variable used as index
29	Imbricated "FOR-NEXT" loops
30	Illegal THEN clause in a "WHEN ... THEN" statement
31	Illegal THEN clause in a "ON IRQ, FIRQ, NMI, ERROR or KEY statement
33	Illegal operation (matrix division)
34	Record length too large in an OPEN statement

....

APPENDIX

D. Runtime Error Messages

0	no error
1	integer division by 0
2	byte division by 0
3	conversion overflow
4	floating-point operation overflow
5	SIN / COS overflow
6	SQR overflow
7	EXP overflow
8	exponentiation (power) overflow
9	LOG / DCLOG overflow
10	ASN / ACS overflow
11	Illegal image
12	string to numeric conversion error
13	computed GOTO / GOSUB index out of range
14	Function key index out of range
15	invalid output logical unit
16	invalid input logical unit
17	illegal input data
18	attempt to read past end-of-data (READ statement)
19	array bounds overflow
20	illegal bit number
21	not enough arguments in calling sequence
22	modulus overflow
23	dynamic array bounds setting error

24	illegal array (MAT IDN)
25	illegal array (MAT INV)
26	attempt to inverse a (almost) singular matrix
27	dimension number error (matrix copy)
28	conversion error or overflow in matrix operation
29	illegal array (scalar operation)
30	illegal array (matrix add / subtract)
31	illegal array (matrix transpose)
32	illegal array (matrix multiplication)
33	hyperbolic function overflow
34	illegal logical unit in an end-of-file test

FATAL RUNTIME ERRORS

129	stack overflow
130	spurious IRQ
131	spurious NMI
132	spurious FIRQ
133	return from main program
134	no disk or operating system not functional
135	illegal or already opened logical unit
136	attempt to open too many logical units at the same time
137	illegal file name (OPEN)
138	no such device
139	device already reserved
140	device not reserved
141	device not ready
142	invalid device
143	duplicate file name

144	file name not found
145	invalid open/closed flag
146	end-of-file
147	invalid file type
148	invalid data transfer type
149	end of media
150	buffer overflow
151	checksum error
152	file is write protected
153	file is delete protected
154	logical sector number out of range
155	no disk file space available
156	no directory space available
157	no segment descriptor space available
158	invalid directory entry number
159	invalid retrieval information block
160	cannot deallocate all space
161	binary record length too large
162	sector buffer size error
163	invalid logical unit (REWIND)
164	SWI executed
165	SWI2 occured
166	SWI3 occured

APPENDIX

E. Summary of BASIC-M Statements and Functions

E.1 Declaration statements

BYTE
INTEGER
DIM
EXTERNAL
DEF

E.2 Input/output statements

INPUT
PRINT
PRINT USING
IMAGE
DATA
READ
RESTORE
OPEN
CLOSE
REWIND

E.3 Control statements

IF ...... THEN
WHEN .... THEN
GO TO
GOTO
GOSUB
ON ...... GOTO
ON ...... GOSUB
CALL
RETURN
FOR .. TO .. STEP
NEXT
ON KEY ..THEN
ON NMI THEN
ON IRQ THEN
ON FIRQ THEN
NEVER KEY ...
NEVER NMI
NEVER IRQ
NEVER FIRQ
NEVER WHEN
STOP
PAUSE

E.4 Matrix statements

```
MAT INPUT
MAT READ
MAT PRINT

MAT V1 = V2 {+,-,*} V3
MAT V1 = V2 {+,-,*,/} ( exp )

MAT V1 = ZER
MAT V1 = IDN
MAT V1 = TRN
MAT V1 = INV
MAT V1 = CON
MAT V1 = SET [ ]
```

E.5 Miscellaneous statements

```
POKE
LINE =
DIGITS =
REM
```

E.6 Built-in functions

```
SIN         ASN         SINH
COS         ACS COSH
TAN         ATN         COTH

EXP         LOG         DCLOG
SQ          SQR
ABS         SGN
INT         MOD
FIX         FLOAT
RND

IAND        IOR         IEOR        ISHFT

LEN         LEFT$       RIGHT$      MID$
TRIM$       ASC         CHR$        STR$
VAL         SUBSTR

PEEK        LOC         TAB         POS
ERR         FKEY        EOF
```

APPENDIX

F. AN812: CHAINING THE EXECUTION OF OVERLAYS

CHAINING THE EXECUTION OF DISK RESIDENT BASIC-M PROGRAM OVERLAYS

Prepared by
Herve Tireford
and
Patrick Monnerat

The BASIC-M repertory of statements does not make provision for a CHAIN instruction as do a few other BASIC interpreters. This note describes a simple method to implement this function thanks to the XDOS system call .COMND. As this system call is available in the XDOS Disk Operating System only, the following description applies to the EXORset BASIC-M interactive compiler.

GENERAL

The general idea behind a CHAIN statement is to partition a large BASIC-M program to run in a disk environment into several modules (or overlays) to be subsequently loaded and executed when needed. This technique, although resulting in some speed degradation due to the overlay loading, is primarily intended to minimize the memory requirements, as there is only one overlay loaded in memory at execution time.

OVERLAYS

Overlays are separately compiled BASIC-M programs. The first overlay loaded in memory must include the BASIC-M Runtime package, if not resident in ROM. The subsequent overlays, however, need not include the Runtime, and therefore consist of the user code only, provided they were compiled with the "R" option specifying the same Runtime start address as the first overlay.

DATA PRESERVATION

Data (variables) which are to be shared by the overlays must be defined in a common memory area; this is usually done by declaring the variables with an address assignment specification (ADDRESS clause). This type of variables is not initialized by the Runtime package, so no assumption must be made as to their initial value. In addition, for XDOS 3.0, the *common variables area must not start below $2200* when using the method described. For XDOS 4.0, the lowest origin for the common variables is $2400.

An alternate solution for passing variables from one overlay to another is to save them in a disk file on completion of the current overlay and to retrieve them on execution of the next overlay.

When establishing a program memory map, take into account that *60 bytes* following the last memory address loaded are used during the overlay load process, and the *last 200 bytes* at the top of available RAM are used by XDOS.

CHAIN PRINCIPLE

Upon termination of the current overlay, an external procedure CHAIN is initiated; actually, this procedure consists in initializing the MC6809 X-register to point to a buffer containing the name of the next overlay to load/execute, prior to executing the XDOS system call .COMND. As a result, and instead of re-entering the XDOS command interpreter, the overlay whose name is stored in the buffer pointed to by the X-register will be loaded and brought to execution. The reader is encouraged to refer to the XDOS User's Guide (paragraph 20.5.6) for the complete description of the .COMND System Call.

The user-supplied CHAIN procedure consists of the following:

	instruction		code
CHAIN	LDX	#BUFFER	8E XX XX
	SCALL	.COMND	3F 40

It may be defined in an assembly language module; due to its short size, however, it can be more easily supplied as part of the BASIC-M program as shown below:

```
00010   BYTE CALLOV(5)
00020   EXTERNAL CHAIN ADDR CALLOV
00030   CALLOV(1)=$8E                                 \ LDX #
00040   CALLOV(2)=ISHFT ((LOC(C$)+1),-8)              \ MSB of ADDR(C$+1)
00050   CALLOV(3)=IAND ((LOC(C$)+1),$FF)              \ LSB of ADDR(C$+1)
00060   CALLOV(4)=$3F                                 \ SWI
00070   CALLOV(5)=$40                                 \ .COMND
            :
            :
00500   C$="OVJ.LO"+CHR$($D)          \  initialize buffer for CHAIN
00510   CHAIN                         \  call overlay OVJ.LO
```

The sample program shown in the appendix uses another method for supplying the CHAIN procedure code from the BASIC-M program, based on the DATA and MAT READ statements. Whichever method is used, the user is cautioned that the X-register is to point to the *first ASCII character* of the buffer, whose address is given by the function LOC(C$)+1. In effect, it should be remembered that the first byte of a string C$ (the one at address LOC(C$)) actually contains the string length, and not the first ASCII character of the string!

The BASIC-M program above equates the CHAIN procedure with the 5-byte table CALLOV(5) (lines 10 and 20); in other words, when control is transferred to the CHAIN procedure, the MPU actually begins to execute the sequence of code contained in this table. This code is stored in the table during execution of lines 30 thru 70. Lines 40 and 50 store the most significant and least significant, respectively, byte of the address +1 of the buffer C$ in the second and third location of the table CALLOV. Line 500 merely initializes the command buffer C$ with the name of the overlay to execute next, terminated by carriage return. Of course, this implies that a file OVJ.LO be found in the disk directory at execution time. Should this file not be found, the message "WHAT ?" will be displayed.

EXAMPLE

The appendix illustrates the method just described. A BASIC-M program has been partitioned in three source overlays OV0, OV1, and OV2, all starting at the same address $3000, and all using the same data section based at $2500. They all assume that the runtime originates at $6500. In this example, the sole variable common to the three overlays is a vector A(5) which is based at $2200 (lowest address for a common section).

- .OV0 is the first overlay; it reads in 5 numeric values from the keyboard and stores them in the vector A(5). It also requests a string variable DO$ which may assume the value "SIN" or the value "COS"; depending on DO$, OV0 will invoke either OVSIN.LO (the object file corresponding to the source overlay OV1) or OVCOS.LO (the object file corresponding to the source overlay OV2).
- .OVSIN.LO stores in a disk file names RESULT each element of A(5) and its sine value.
- .OVCOS.LO stores in RESULT each element of A(5) and its cosine value.
- .OVSIN.LO and OVCOS.LO each chains the execution of the XDOS LIST command to list the file RESULT they constructed. This implies that the LIST command be available on the disk in drive 0. On completion of LIST, XDOS is re-entered.

The listing in the appendix shows the different steps to be followed. Note the use of the compiler-mode (option "O" is specified when invoking BASIC-M) to return the user-code directly to the disk. Also note that OV1.LO and OV2.LO are renamed OVSIN.LO and OVCOS.LO, respectively, to be compatible with the names of the overlays which can be called from the first overlay OV0.LO.

APPENDIX

```
=
=BASICM OV0;0


              BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979


READY
LIST

00010   BYTE CALLOV(5)
00020   EXT CHAIN ADDRESS CALLOV
00030   DIM C$(1) ADDR $50
00040   DATA $8E,$0,$51,$3F,$40
00050   MAT  READ CALLOV
00060   REM
00070   DIM A(5) ADDR $2200
00080   MAT  INPUT "enter A(5) : ",A
00090   INPUT "enter SIN or COS : ",DO$
00100   IF  NOT(DO$="SIN" OR DO$="COS") THEN 90
00110   C$="OV"+DO$+".LO"
00120   PRINT "CALLING OVERLAY ... ",C$
00130   C$=C$+ CHR$($D)
00140   CHAIN

READY
COMPILE M,R=$6500,D=$2500

NO ERROR

CALLOV....................B......2500....1
CHAIN.....................E......2500.....
C$........................S......0050....1
A.........................R......2200....1
DO$.......................S......250B.....

DSCT: 2500-29DA
PSCT: 6D98-6F21

READY
QUIT
CREATE OBJECT FILE OV0      .LO:0 (Y/N) ? Y

ENTER PROGRAM HEX ORIGIN ($XXXX) : $3000
```

```
=BASICM OV1;0


             BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979


READY
LIST

00010  BYTE CALLOV(5)
00020  EXT CHAIN ADDRESS CALLOV
00030  DIM C$(1) ADDR $50
00040  DATA $8E,$0,$51,$3F,$40
00050  MAT  READ CALLOV
00060  REM
00070  DIM A(5) ADDR $2200
00080  REM
00090  OPEN #3,"RESULT",U
00100  REWIND #3
00110  FOR I=1 TO 5
00120  PRINT #3  USING 130,A(I), SIN(A(I))
00130  IMAGE "A(I)= [2,5][X10]SIN(A(I))= [2,7]"
00140  NEXT I
00150  PRINT #3 "END OF FILE RESULT"
00160  CLOSE #3
00170  C$="LIST RESULT"+ CHR$($D)
00180  CHAIN

READY
COMPILE M,R=$6500,D=$2500

NO ERROR

CALLOV...................B......2500....1
CHAIN....................E......2500.....
C$.......................S......0050....1
A........................R......2200....1
I........................R......250B.....

DSCT: 2500-29BF
PSCT: 6DBE-6FB4

READY
QUIT
CREATE OBJECT FILE OV1      .LO:0 (Y/N) ? Y

ENTER PROGRAM HEX ORIGIN ($XXXX) : $3000
```

PARTITIONING A BASIC-M SOURCE PROGRAM

Prepared by
Herve Tireford
and
Patrick Monnerat

BASIC-M source programs may be such that their size or memory requirements render their compilation impossible due to the BASIC-M compiler design approach which assumes the source be wholly memory-resident at the time compilation is initiated. There are several methods which can be used separately or jointly to overcome this problem: use of the compiler-mode, use of the compiler "S" option to minimize the object code requirements, assignment of the Data Section, coding of constants as hexadecimal values, definition of integer or byte variables whenever possible, partitioning of the source into several modules to be compiled separately and chained at execution using the XDOS SCALL .CMND, . . . etc. This note describes how to partition the source into several modules which are compiled separately, and which may reside in ROMs in the final environment. It outlines the user-program design constraints, and illustrates the assembly routine used to call one module from another. We are restricting this study to a two-object module partition.

COMPILER CODE GENERATION

The following code is generated by the BASIC-M compiler at the beginning of each object program:

```
START  CLRA
       LDS    #STACK   Stack pointer and data section initialization
       JSR    INIT
       FCC    /VVRR/   Runtime version/revision
       FDB    DSEC     Start address of data section
       FDB    PSEC-START Offset to statement code
            :
       DATA constants and array descriptors
            :
       FCB    0
PSEC   EQU    *        Beginning of statement code
            :
```

PARTITIONING THE SOURCE PROGRAM

Let's assume that the source needs to be partitioned in two modules, hereafter referred to as M1, and M2. M1 is the main module, i.e., it contains the object code to which control is transferred first. The following rules apply:

1. M2 must be written as a subroutine and therefore must terminate with a RETURN statement, unless control is not given back to M1.
2. The variables local to M1 and those local to M2 must reside in two distinct data sections, the origins of which are specified in the COMPILE command. Of course, the user must insure that the two data sections do not overlap. To that end, it is recommended to compile M1 first, and then to deduce the origin of the data section for M2 from the highest data section address of M1 as reflected in the symbol table issued on completion of the compilation of M1.
3. The global variables, i.e., those common to M1 and M2, must be explicitly defined in each module by a declaration statement to assign the variable absolute address (ADDRESS clause). It should be emphasized that such variables will not be initialized by the runtime package, therefore no assumption must be made as to their initial value.
4. All the DATA statements must reside in M1.
5. In order to obtain an accurate indication of error in the event one occurs, it is recommended (but not mandatory) that line numbers in M1 be distinct from line numbers in M2.
6. M1 statements cannot transfer control to a specific statement in M2, and vice-versa. It is only possible to call a secondary module (M2) from another module.
7. M1 cannot call user-written functions/procedures defined in M2, nor can M2 call functions/procedures defined in M1.
8. In order to transfer control to M2 from M1, an external assembly procedure, hereafter referred to as "CALLM2", needs to be declared in M1, and further activated when desired.
9. Statements which may implicitly transfer control from one module to the other must be deactivated prior to entering a given module and reactivated upon return from the called module. Those statements include:
   - .WHEN ... THEN
   - .ON ERROR THEN
   - .ON NMI (IRQ, FIRQ) THEN
   - .ON KEY ... THEN

ASSEMBLY CONTROL ROUTINE "CALLM2"

This subroutine is listed in Figure 1. It supports a real or integer argument which dictates whether the data section of module M2 must be cleared or not upon entry in M2. Note that on the first call to CALLM2 one must specify no argument or an argument equal to zero so as to initialize the data section of M2. Not doing so may preclude the normal recognition of execution errors. Further calls to M2 may specify an argument different from zero if the user desires to preserve the data of M2 as set up by the previous call.

EXAMPLE

The appendix contains a sample program to illustrate the procedures and rules described. A BASIC-M program has been split in two modules M1 and M2. M1 is intended to generate 100 random numbers in a vector A(100). M2 is aimed at printing a subrange of the same vector A between two subscripts K and L to be input at execution time. The example assumes that the BASIC-M runtime package starts at $6500. The MERGE command concatenates the object modules CALLM2 (org $2000), M1 (org $2200), and M2 (org $2800) into the final user code OBJECT, and forces the M1 origin as start address.

```
                        NAM    CALLM2
                        OPT    NOP,LLEN=120
2000                    ORG    $2000

                   * THIS SUBPROGRAM TRANSFERS CONTROL TO MODULE M2
                   * BASICM CALL : CALLM2(ARG)
                   * ARG IS AN OPTIONAL ARGUMENT. IF ARG = 0 OR ARG IS NOT SPECIFIED, THEN
                   * THE DATA SECTION OF MODULE 2 IS CLEARED. IF ARG IS NOT 0, THEN THE DATA
                   * SECTION IS NOT INITIALIZED. THE DATA SECTION OF MODULE 2 MUST BE
                   * CLEARED UPON FIRST CALL TO THIS MODULE.

             2800  START2 EQU  $2800     START ADDRESS OF MODULE M2

             0020  RAMAD  EQU  $20       VALID WITH BASICM 2.02 ONLY !
                   ********************************************************************
                   * IMPORTANT : IN BASICM RELEASES HIGHER THAN 2.02, RAMAD IS DEFINED *
                   * ==========  IN THE FIRST TWO BYTES OF THE RUNTIME PACKAGE.        *
                   ********************************************************************

2000 8E   2800     CALLM2 LDX  #START2
2003 10AE A4              LDY  ,Y        CHECK IF ARGUMENT
2006 27   04   200C       BEQ  L3        NO ARGUMENT. CLEAR DSCT.
2008 EC   A4              LDD  ,Y        ARGUMENT=0 ?
200A 26   0A   2016       BNE  L1        NO. DO NOT CLEAR DATA SECTION
200C 10AE 0C       L3     LDY  12,X      GET DSCT BEGINNING ADDRESS
200F 6F   A0       L2     CLR  ,Y+       CLEAR DSCT
2011 10AC 03              CMPY 3,X       DONE ? (DSCT END ADDRESS IS STACK INIT VALUE)
2014 23   F9   200F       BLS  L2        NOT YET.
2016 EC   9F 0020  L1     LDD  [RAMAD]   SAVE CURRENT PSCT ORIGIN OF CALLING MODULE
201A 34   06              PSHS D
201C AF   9F 0020         STX  [RAMAD]   DEPOSIT PSCT ORIGIN OF MODULE CALLED
2020 EC   0E              LDD  14,X      GET OFFSET TO PROGRAM ORIGIN
2022 AD   8B              JSR  D,X       CALL MODULE
2024 35   06              PULS D         RESTORE PSCT ORIGIN OF CALLING MODULE
2026 ED   9F 0020         STD  [RAMAD]
202A 39                   RTS            PROCEED WITH EXECUTION OF CALLING PROGRAM
                          END
TOTAL ERRORS 00000--00000
TOTAL WARNINGS 00000--00000
```

FIGURE 1 — CALLM2

APPENDIX

```
=BASICM M1;O


              BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979


READY
LIST

00010  REM ----- MODULE M1
00020  REM ----- PSCT BASED AT $2200 *** DSCT BASED AT $600
00030  REM -----
00040  REM ----- GENERATE 100 RANDOM VALUES IN ARRAY A(100)
00050  REM ----- CALL MODULE M2 TO HAVE A SUBRANGE OF A(100)
00060  REM ----- LISTED (FROM ROW K TO ROW L)
00070  REM -----
00080  EXTERNAL CALLM2 ADDR $2000
00090  REM ----- COMMON VARIABLES DECLARATION
00100  REM ----- COMMON SECTION BASED AT $100
00110  INTEGER PASS ADDR $0100
00120  DIM A(100) ADDR $0102
00130  REM -----
00140  DATA "WE ARE NOW IN M2","WE ARE NOW BACK IN M1"
00150  PASS=$0 \ INITIALIZE PASS TO 0
00160  FOR I=1 TO 100
00170  A(I)= RND
00180  NEXT I
00190  PASS=PASS+1
00200  CALLM2(PASS-1) \ ON FIRST CALL, DATA SECTION WILL BE CLEARED
00210  READ MESG$
00220  PRINT MESG$
00230  RESTORE
00240  GOTO 160

READY
COMPILE M,R=$6500,D=$600

NO ERROR

CALLM2....................E......2000.....
PASS......................I......0100.....
A.........................R......0102....1
I.........................R......0602.....
MESG$.....................S......0607.....

DSCT: 0600-0AD6    ...So let's start M2 DSCT at $B00.
PSCT: 6EDA-7048

READY
QUIT
CREATE OBJECT FILE M1       .LO:0 (Y/N) ? Y

ENTER PROGRAM HEX ORIGIN ($XXXX) : $2200
=
```

```
=BASICM OV2;O


            BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979


READY
LIST

00010  BYTE CALLOV(5)
00020  EXT CHAIN ADDRESS CALLOV
00030  DIM C$(1) ADDR $50
00040  DATA $8E,$0,$51,$3F,$40
00050  MAT  READ CALLOV
00060  REM
00070  DIM A(5) ADDR $2200
00080  REM
00090  OPEN #3,"RESULT",U
00100  REWIND #3
00110  FOR I=1 TO 5
00120  PRINT #3  USING 130,A(I), COS(A(I))
00130  IMAGE "A(I)= [2,5][X10]COS(A(I))= [2,7]"
00140  NEXT I
00150  PRINT #3 "END OF FILE RESULT"
00160  CLOSE #3
00170  C$="LIST RESULT"+ CHR$($D)
00180  CHAIN

READY
COMPILE M,R=$6500,D=$2500

NO ERROR

CALLOV...................B......2500....1
CHAIN....................E......2500.....
C$.......................S......0050....1
A........................R......2200....1
I........................R......250B.....

DSCT: 2500-29BF
PSCT: 6DBE-6FB4

READY
QUIT
CREATE OBJECT FILE OV2      .LO:0 (Y/N) ? Y

ENTER PROGRAM HEX ORIGIN ($XXXX) : $3000
```

```
=NAME OV1.LO,OVSIN.LO
=NAME OV2.LO,OVCOS.LO
=BASICM DUMMY

            BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979



READY
QUIT
SAVE (Y/N) ?N
=OV0.LO
enter A(5) : ? 3.14159265  1.57  0  1  2
enter SIN or COS : ? TAN
enter SIN or COS : ? COS
CALLING OVERLAY ... OVCOS.LO
```

This, just to load the Runtime at $6500 (default load address in BASIC-M)

load/execute first overlay

(What follows is the result of the execution of the "LIST RESULT" command invoked from OVCOS.LO)

```
PAGE 001   RESULT   .SA:0

A(I)=  3.14159          COS(A(I))= -0.9999999
A(I)=  1.57000          COS(A(I))=  0.0007963
A(I)=  0.00000          COS(A(I))=  0.9999999
A(I)=  1.00000          COS(A(I))=  0.5403023
A(I)=  2.00000          COS(A(I))= -0.4161468
END OF FILE RESULT
```

```
=BASICM M2;O

              BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979


READY
LIST

01000  REM ------ MODULE 2
01010  REM ------ PSCT BASED AT $2800 *** DSCT BASED AT $B00
01020  REM ------
01030  REM ------ IT PRINTS OUT A SUBRANGE OF ARRAY A(100)
01040  REM ------ DECLARE COMMON VARIABLES
01050  INTEGER PASS ADDR $0100
01060  DIM A(100) ADDR $0102
01070  REM ------
01080  READ M$
01090  PRINT M$
01100  PRINT  USING 1110,PASS
01110  IMAGE "THIS IS PASS #[2]"
01120  INPUT "SUBRANGE K AND L : ",K,L
01130  IF K>L THEN 1120
01140  FOR INDEX=K TO L
01150  PRINT INDEX,A(INDEX)
01160  NEXT INDEX
01170  RETURN

READY
COMPILE M,R=$6500,D=$B00

NO ERROR

PASS....................I......0100.....
A.......................R......0102....1
M$......................S......0B02.....
K.......................R......0B22.....
L.......................R......0B27.....
INDEX...................R......0B2C.....

DSCT: 0B00-0FE0
PSCT: 6DEF-6F44

READY
QUIT
CREATE OBJECT FILE M2       .LO:0 (Y/N) ? Y

ENTER PROGRAM HEX ORIGIN ($XXXX) : $2800
=
```

```
=MERGE CALLM2.LO,M1.LO,M2.LO,OBJECT.LO;2200
=BASICM DUMMY


            BASIC-M 2.02
 COPYRIGHT BY MOTOROLA, INC. 1979


READY
QUIT
SAVE (Y/N) ?N
=LOAD OBJECT

.;P WE ARE NOW IN M2
THIS IS PASS # 1
SUBRANGE K AND L : ? 2 4
2                          9.15583223E-05
3                          6.40887301E-04
4                          3.11284885E-03
WE ARE NOW BACK IN M1
WE ARE NOW IN M2
THIS IS PASS # 2
SUBRANGE K AND L : ? 99 101
99                         0.674710883
100                        0.876549642

***        ERROR # 19 AT LINE  1150
101                        -4151162.5
WE ARE NOW BACK IN M1
WE ARE NOW IN M2
THIS IS PASS # 3
SUBRANGE K AND L : ? 1 0
SUBRANGE K AND L : ? 2 2
2                          0.932374047
WE ARE NOW BACK IN M1
WE ARE NOW IN M2
THIS IS PASS # 4
SUBRANGE K AND L : ?
STOP ***   OPERATOR ABORT    ***

.
```

This is just for loading the Runtime at $6500 (default address)

(index out of range)

(CTRL-P was typed)

APPENDIX H

SUPPLEMENTARY MANUAL

KERNEL REQUIREMENTS

1.1. INTRODUCTION

This manual is a supplement to the M6809 BASIC-M User's Guide; it discusses the requirements for a user-written firmware which allows the object code produced by the EXORset BASIC-M compiler (release 3.00 and higher) to run in conjunction with the EXORset BASIC-M Runtime package, in an M6809-based end user system.

A complete example is presented in appendix B that outlines the procedures to be followed to develop a specific application firmware to run in a Micromodule environment (M68MM19).

1.2. BASIC-M SOFTWARE ENVIRONMENT

Three levels of program code are needed for the execution of a BASIC-M program:

.LEVEL A: Highest level consisting in the object code produced by the BASIC-M compiler; this is the true user program; it is hereafter referred to as the "user code".

.LEVEL B: The Runtime package code.

.LEVEL C: The lowest level routines which form what we will call the "kernel". The kernel is the only object code which is system-dependent; it contains the I/O routines, interrupt handlers, and so forth.

Level A code interfaces with Level B subroutines, which in turn, interface with Level C code, as is represented in figure 1.1.

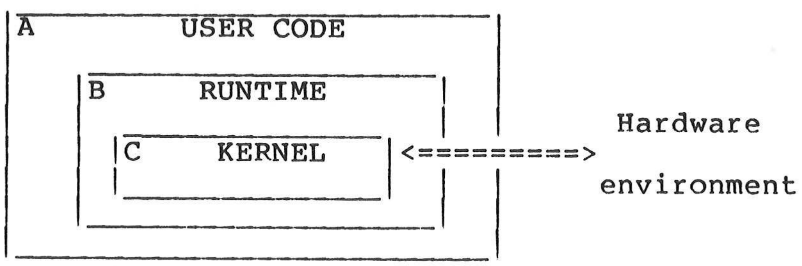

Figure 1.1. BASIC-M Software Environment

1.3. USER AND RUNTIME CODE CHARACTERISTICS

The User and Runtime Codes are both ROMable and position-independent. Position independency implies that these codes can be physically located anywhere among the 64K memory map. However, the user code assumes that the Runtime Package is installed at the same address as the one specified at compilation time. The Runtime and User Code origins, as well as the origin of the data section can be specified at the time BASICM is invoked in the "compiler" mode. Of course, there must be no hardware conflict (no overlap) between the following sections :

.The Kernel,
.The Runtime Package,
.The User Code,
.The RAM Data Section.

The kernel installation requirements are detailed in the next paragraph. The Runtime Package origin and RAM Data Section starting address are supplied in the BASICM invoking command ("compiler" mode); for instance, the following command generates a user code that will execute in a target system where the Runtime Package is installed at hex base address 8000 and where RAM exists from E000 onwards.

```
=BASICM SAMPLE;OL=#LP,R=$8000,D=$E000
```

The memory sizes of the user code and RAM Data Section are shown in the symbol table printout.
The size of the Runtime Package required to run the User Code varies between 10K and 14K bytes depending on the BASIC-M source program (see paragraph 1.8).

1.4. THE KERNEL FUNCTION

The task of the user-written kernel is to provide the hardware/software interface. The functions it must fulfill can be split in four categories:

a. Start-up routine

This routine is given control at the time of the system start-up or restart. It must initialize the system peripherals and pass control over to the User Code (level A).

b. I/O routines

They are the standard peripheral device drivers, and are accessed each time a BASIC-M I/O statement is encountered in the course of program execution, or when an error message is output by the Runtime Package.

c. Interrupt control

The kernel must include the interrupt vectors and the interrupt primary handlers.

d. Closeout routine

This is the portion of code which is executed when a "STOP" or "END" statement is encountered, or upon completion of the user code. It may halt the processor (CWAI instruction), or start another process.

1.5. THE KERNEL SPECIFICATIONS

The following rules must be observed to insure compatibility between the kernel and the Runtime Package.

.Rule A: the kernel must respond to the address range F000 thru FFFF, even though it does not fully occupy this address space.

.Rule B: the RESTART vector points to the kernel start-up routine.

.Rule C: the start-up routine must initialize the standard peripherals that may be attached to the user system (console, printer); this is not done by the Runtime Package, nor by the User Code.

.Rule D: upon completion, the start-up routine must transfer control to the first byte of User Code (level A).

.Rule E: Interrupt handling.

The highest locations of the kernel, i.e those responding to the address range FFF2 to FFFF, must contain the restart and interrupt vectors. The RESTART vector has already been discussed (Rule B). The other vectors must point to primary interrupt handlers which themselves must transfer control to secondary interrupt handlers (Runtime-resident) whose addresses are maintained in a table in RAM data section. The pointer to the top address of the secondary interrupt vector table is held in a 16-bit RAM location whose address is to be specified in the kernel locations FFEE, FFEF. This pointer is controlled by the Runtime Package; further to the execution of a STOP or END statement, the pointer is reset to zero; therefore, interrupts that may occur afterwards should be handled separately; usually the user should treat them as spurious (undesirable) interrupts.

Figure 1.2 illustrates the indirect interrupt vectoring scheme just described; figure 1.3 contains a listing that shows how the kernel must be structured as far as interrupt handling is concerned.

.Rule F: Input/Output drivers.

Two standard peripheral devices are treated by the Runtime package: the console and the line printer. This implies the definition of three functions:

.input from console,
.output to console,
.output to line printer.

The line printer output driver is only required if the BASIC-M source program includes a statement of the type PRINT #LU or MAT PRINT #LU where LU (logical unit specification) is equated to 2.

The console input driver is required if the source program contains one of the following statements :

INPUT, MAT INPUT, INPUT #LU, MAT INPUT #LU, or PAUSE, where LU is equated to 0 or 1.

The console output driver, however, must be provided anyway, since the error processing, the STOP, PAUSE, and END statements may access it any time.

The driver entry points and functions must be provided in the kernel as follows:

INCHNP	$F015	Input a character from the console to the A register (most significant bit must be cleared) and echo it. The other registers must be preserved. Optional entry point.
OUTCH	$F018	Output the character in the A register to the console display device. The registers must be preserved. Mandatory entry point.
PDATA	$F024	Output to the console display device a carriage return and line feed characters followed by the character string pointed to by the X register and terminated with an EOT character (04). Mandatory entry point.
LIST	$F042	Output the character in the A register

to the line printer. The registers must be preserved. If successful, return with carry bit clear, if not, set carry bit upon return. Optional entry point.

In addition, a break condition test routine must be provided if the BASIC-M source program has not been compiled with the "S" option, or if the BASICM program performs line printer output or matrix operations.

CKBRK $F045 Test a break condition; return with carry clear if no break, with carry set if the user program is to be aborted. The A register can be altered; the other registers must be preserved.

I M P O R T A N T   N O T E

Never modify the stack pointers U and S other than with PSH and PUL instructions.

.Rule G: Closeout routine.

The entry to this routine is performed upon execution of the STOP or END statements, or upon program completion. This routine may not return.

EXIT $F02D Terminates the program execution.

1.6. RAM STORAGE

The starting address of the RAM DSCT section which is used by the program variables and stacks, is memorized by the Runtime Package in a 16-bit RAM word referred to as the Data Section Link (DSCT LINK). The address of the DSCT LINK is held in the first two bytes of the Runtime Package. The "as-delivered" address of the DSCT LINK for the EXORset BASICM Runtime Package is $0020; it can be changed, if desired. The BLOAD utility which is supplied on each BASICM diskette provides a convenient means to change this value (see paragraph 1.9). The DSCT LINK is controlled by the Runtime and MUST not be altered during the execution of the program.

1.7. FUNCTION KEYS

A function keys interrupt handler is included in the Runtime Package. This handler is accessed on every NMI interrupt which occurs when keys interrupts are enabled further to the execution of "ON KEY..." statements. Should the user desire to take advantage of function keys monitoring, he must provide in his system a hardware circuitry which is fully compatible with the one in EXORset 30. Refer to the EXORset User's Guide for a complete description.

1.8. RUNTIME PACKAGE SIZE

BASIC-M programs which do not include matrix-oriented statements (those starting with the "MAT" keyword) nor disk I/O statements require that the first 10K bytes bytes of the Runtime Package be resident at execution time, whereas the full 14K bytes are needed to cope with matrix operations.

Using the BLOAD utility provides an easy means to tailor the Runtime Package to the application requirements.

1.9. BLOAD UTILITY

The BLOAD command available on the EXORset BASICM diskette allows to merge multiple files comprising the User Code file, assembly language-written files and all or part of the Runtime Package into a single file which represents the application firmware that runs in conjunction with the user-supplied kernel. In addition, BLOAD produces a memory map showing the boundaries of each module loaded. The BLOAD functions and options are described in the BASCNEWS file of a BASICM diskette. Below is just an example of the construction of a file TEST.CM that is the concatenation of a User cod e file TEST.LO (generated by the BASICM compiler) that is origined at $A800, and of the Runtime package without matrix nor disk routines. The Runtime package has been specified at compile time to origin at $8000 in the end-user system. The DSCT LINK (see paragraph 1.6.) is located at address $E7FC.

```
=BLOAD TEST;ORL=$E7FC,-M
TEST    .LO:0  A800 - A8A7
RUN-TIME       8000 - A7A6
DSCT LINK      E7FC - E7FD
DATA SECTION   E000 - E4B4

LOAD BOUND S    8000 - A8A7
=
```

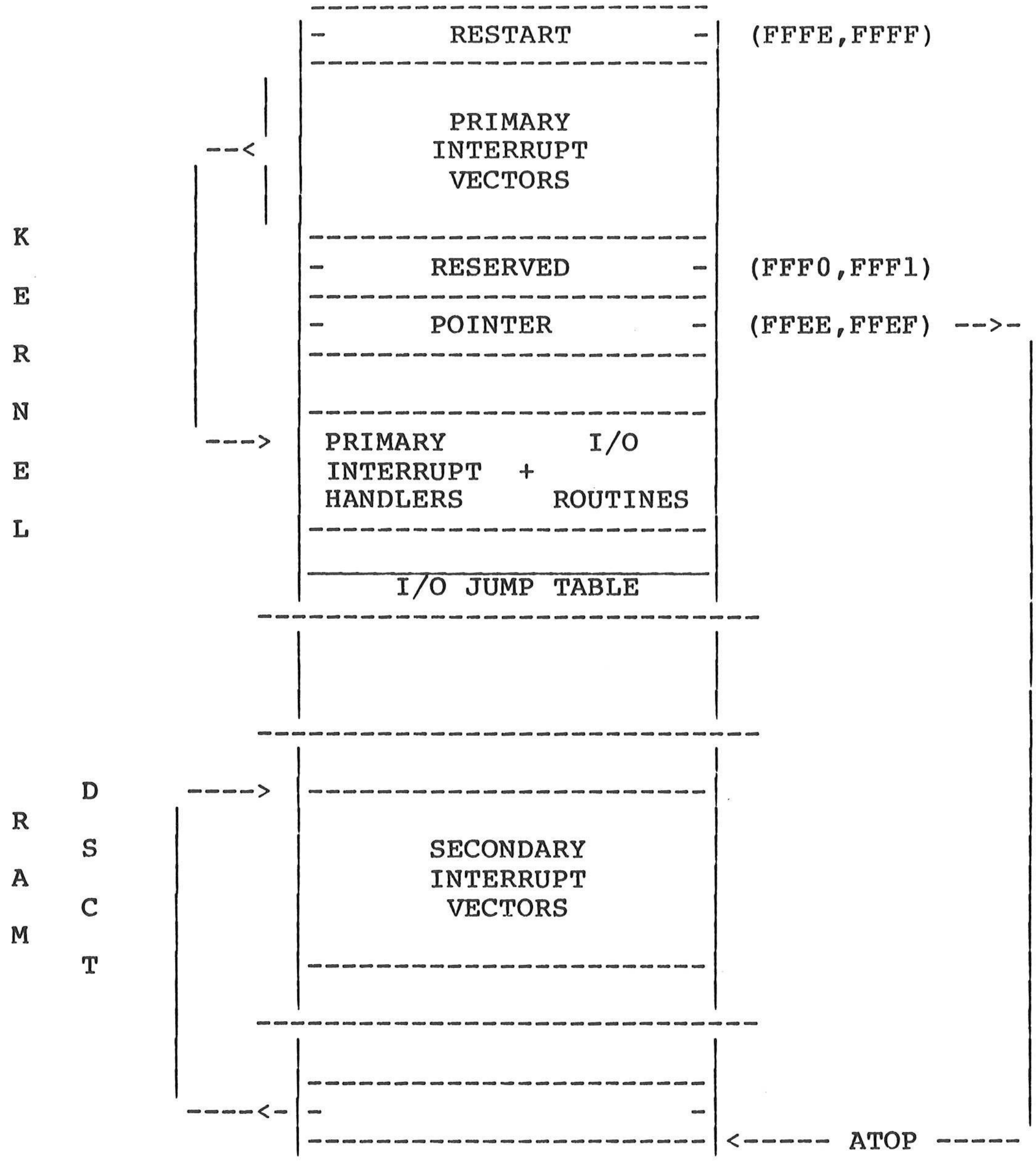

Figure 1.2. Interrupt handling

The primary interrupt handlers transfer control to the Runtime secondary handlers by accessing the secondary interrupt vectors contained in a table pointed to by the pointer whose address is specified in FFEE, FFEF. The pointer and secondary interrupt vectors contents are set up by the Runtime Package upon entry in the BASIC-M program (user code).

```
* INTERRUPT VECTORS
*
         ORG    $FFEE
         FDB    ATOP         ADDRESS OF POINTER TO SECONDARY VECTORS TABLE
         FDB    -1           RESERVED
         FDB    SWI3
         FDB    SWI2
         FDB    FIRQ
         FDB    IRQ
         FDB    SWI
         FDB    NMI
         FDB    RESTAR       ADDRESS OF STARTUP ROUTINE
*
ATOP     EQU    $E72E        MUST BE IN RAM
*

         ORG    $XXXX        INTERRUPT ROUTINES IN KERNEL

* PRIMARY INTERRUPT HANDLERS
*
NMI      BSR    INTER        NMI SERVICE ROUTINE
         FCB    -3           MSB OF VECTOR IS A POINTER -3
SWI      BSR    INTER        SWI SERVICE ROUTINE
         FCB    -5
IRQ      BSR    INTER        IRQ SERVICE ROUTINE
         FCB    -7
FIRQ     BSR    INTER        FIRQ SERVICE ROUTINE
         FCB    -9
SWI2     BSR    INTER        SWI2 SERVICE ROUTINE
         FCB    -11
SWI3     BSR    INTER        SWI3 SERVICE ROUTINE
         FCB    -13
*
* CENTRAL PRIMARY INTERRUPT HANDLER
*
INTER    PSHS   X,B,CC       SAVE REGISTERS USED IN THIS ROUTINE
         LDX    >ATOP        FETCH SECONDARY TABLE TOP ADDRESS
         BEQ    NOVEC        NOT SPECIFIED. ERROR.
         LDB    [4,S]        FETCH VECTOR OFFSET
         LDX    B,X          FETCH CORRESPONDING SECONDARY HANDLER ADDRESS
         BEQ    NOVEC        ILLEGAL ADDRESS
         STX    4,S          MODIFY RETURN ADDRESS ON SYSTEM STACK
         PULS   X,CC,B,PC    RESTORE REGISTERS AND GO TO INTERRUPT ROUTINE
*
NOVEC    ...                 VECTOR ERROR PROCESSING
*
```

Figure 1.3. Listing of the kernel interrupt routines
---------- and vectors structure.

CHAPTER 2. KERNEL FOR MICROMODULE M68MM19

2.1. INTRODUCTION

This chapter describes a general-purpose kernel that has been specifically written to interface a BASIC-M application software to an end-user system based on micromodule M68MM19, thereafter referred to as MM19. The proposed kernel makes some assumptions as to the usage of the MM19 I/O adapters and memory map; for instance, the ACIA port is used as the system console port, thus requiring that a terminal be connected to it, and the PIA port is used to interface to a Centronics-type line printer. In addition, the kernel also offers facilities to load into the system RAM the XDOS 4 disk operating system, and possibly the application software; therefore, disk I/O operations under control of the application program are feasible, with the restriction that no interrupt must occur during disk operations.

The features incorporated in the kernel are likely to satisfy most of the users needs. However, there are certainly applications requiring a hardware environment different from the one implied by the proposed kernel; for those, the kernel listed in Appendix A should still provide a valuable aid as a starting point for an adaptation.

2.2. HARDWARE ENVIRONMENT

The following environment is assumed by the proposed kernel:

.kernel based at $F000,
.top of addressable memory at $FFFF,
.console ACIA based at $EC14,
.line printer PIA based at $EC10,
.2K bytes of RAM based at $E000 (can be used to hold the Data section of the application program).

If disk I/O operations are required, the user shall install in his system the EXORset mini-floppy disk controller or the EXORdisk III floppy-disk controller equipped with the appropriate EXORset disk driver. Either of these boards occupies memory space between $E800 and $EC0F; if the EXORset mini-floppy disk controller is used, the on-board 16K RAM must be disabled.
In addition, the system must include the amount of contiguous RAM necessary to hold the XDOS 4 disk operating system and the application program, should it not be resident in (E)ROM´s.

2.3. THEORY OF OPERATIONS

This section gives some explanations on the operations of the proposed kernel listed in Appendix A.

2.3.1. Start-Up

Upon system start-up or restart, the console ACIA and the line printer PIA are initialized, and a 16-bit word is retrieved from the first two locations of the kernel labelled APSTRT; if the word just retrieved is not zero, it is supposed to be the starting address of the application program; control is then passed over to it. If APSTRT contains a 16-bit null value, the kernel loads the XDOS 4 disk operating system by entering the disk driver at $E800. In case no disk is available (ready) in drive 0, the user is prompted with the message:

INSERT DISK IN DRIVE 0

An automatic disk bootstrap is performed as soon as the disk is ready; upon its completion, the XDOS 4 prompt sign "=" is displayed at the system console and the user may enter a command of its choice; generally it will be a command to load and execute the application program.

The start-up process is summarized in the flowchart of figure 2.1.

```
                         .-------------.
                         || START-UP  ||
                          -----.------
                               |
                     __________|___________
                    ! I/O Initialization  !
                     ----------.----------
                               |
                        _______|_________
                       !  X <-- APSTRT  !
                        -------.--------

           no                               yes
       .-------<-----       X=0       ----->--.
       |                     ?
 ______|____________
| Enter program at X |              yes    Disk    no
 --------------------            .-----   Ready   -----.
                                 |          ?           v
                                 |                   ___|____
                                 v                ! Wait until !
                                 |                ! disk ready !
                                 |                 -----.------
                                 |______________________|
                                            |
                                     _______|______
                                    | Load XDOS 4 |
                                     -------------
```

Figure 2.1. Kernel Start-Up Process

2.3.2. Kernel Jump Table

The kernel jump table includes all the entry points and routines necessary to the execution of a BASIC-M program that are described in chapter 1. It is derived from the EXORbug monitor jump table; entry points which could be accessed from assembly-language routines, and which would transfer control to utilities not implemented in the proposed kernel all direct the execution to a common routine that, first, displays the address of the routine the user attempted to call, and second, enters the EXIT routine.

2.3.3. EXIT Routine

This routine is entered upon termination of the BASICM application program (STOP, END statements or physical end of program are encountered), or on occurence of a fatal error or operator abort.

If the application program came to its normal completion, the EXIT routine will transfer control to a user-supplied sequence whose address is found in the third and fourth bytes of the kernel (ARSTRT); this routine may for example just restart the application program.

If the application program has not been entered yet, the EXIT routine will prompt the user with the message:

PRESS RESET TO RESTART

and wait idle until the system is restarted.

2.3.4. Other

The other routines of the kernel do not call for special comment as they conform to the specifications described in chapter 1.

The RAM locations labelled ATOP, RAMAD, AECHO, STACK in the kernel (assembly listing lines 71 thru 74) may be assigned addresses other than those listed; however, if XDOS 4 has to be loaded in the system, the definitions of ATOP, AECHO, and STACK must no be changed.

2.3.5. More On Disk I/O

If XDOS 4 is used in the end-user system, the parameters which normally reflect the alternate map disk controller (see XDOS 4 User's Guide) must all be cleared. This is achieved by setting to zero the six values in the Disk Identification Block (PSN 0000) starting at offset $76.

The start-up routine detailed under 2.3.1 describes how the user may optionally load the disk operating system upon system start-up. This process ends up with XDOS 4 displaying its prompt character ("=") and awaiting a system command from the console.

A simple modification to XDOS 4 allows to automatically load and execute a user-selectable command (the application program) instead of entering the normal XDOS 4 command interpreter after XDOS has been loaded. The modification is as follows?:

1/.Concatenate with the file "XDOS.SY" a data file which initializes the XDOS command buffer (CBUFF$ EQU $AE) with the name of the user-selected program terminated by carriage return.

2/.Change the start address of the file XDOS.SY to its current start address +2. This may be accomplished by using the following procedures:

(a) Build and assemble the file containing the name of the command to execute on start-up. This name must be loaded from CBUFF$ upwards and must be terminated with a carriage return character. For the sake of explanation, the file which contains the name of the command to execute on cold-start is referred to as NAMECMD.LO, whereas the user-command file associated is referred to as USERCMD.CM.

(b) Unprotect file XDOS.SY by using:
=NAME XDOS.SY;NX

(c) Read the XDOS.SY normal start address in RIB by using:
=DUMP XDOS.SY
:S FFFF
The XDOS start address is found at offset 7A, 7B. Let it be YYYY.

(d) Concatenate XDOS.SY and NAMECMD.LO, giving the new XDOS.SY named XDOS1.SY. Use the following command where WWWW is equal to YYYY+2 (see step (c) above):
=MERGE XDOS.SY,NAMECMD.LO,XDOS1.SY;WWWW

(e) Rename XDOS.SY by using:
=NAME XDOS.SY,XDOS2

(f) Rename XDOS1.SY to become XDOS.SY, by using:
=NAME XDOS1.SY,XDOS.SY;WDS

(g) Generate a new diskette in drive #1, by using:
=BACKUP ;R
Select the files you want to have copied (the user command USERCMD.CM must be selected !).
The "BACKUP ;R" command insures that the XDOS.SY load and start addresses are written into sector 18 of the diskette.

(h) Restore original XDOS.SY on drive #0, by using:
=NAME XDOS.SY,XDOS1;NX
=NAME XDOS2.SY,XDOS.SY;WDS

The new diskette thus generated in drive #1 is then ready for autostart.

CHAPTER 3. APPLICATION EXAMPLE

This chapter describes an application based on MM19. For the sake of simplicity, the application is kept simple even if it may look somewhat academic; the aim is more to detail the procedures that lead to the execution of the application program in a micromodule environment than to detail the program itself.

3.1. APPLICATION DESCRIPTION

The MM19 is used as a computer whose function is to find how to combine six numbers in order to yield a given target number by applying elementary arithmetic operations (*,+,-). The seven numbers (the first six and the target) are input from the console; for example, if MM19 is given the 6 numbers

5 25 4 75 6 5

and the target number 469, the application program must be able to determine that the target can be obtained by applying the following successive operations:

```
  4  x  25  = 100
100  -   5  =  95
 95  x   5  = 475
475  -   6  = 469
```

As is shown on this example, the solution need not use all the six input numbers; however, a number can be used only once.

Of course, there may be several solutions to a problem; the program is required to output only one (that may not be the most efficient !). Also, there may be no exact solution; in that case, the program will report it.

Optionally, the seven numbers may be generated randomly instead of being input from the console. In any event, the target number is included between 100 and 999, and the other six numbers can be any of the following:

1..9 , 10 , 25 , 50 , 75 , 100

The problem and its solution can be at will listed to the system console or line printer. To have more fun, the MM19 is requested to find a solution in a fixed time frame of one minute; if a solution is found before this time elapsed, the program will report the time it took to come to it.

A sample run of the program will help to understand how the program works (see 3.4. now ... or later on).

3.2. HARDWARE ENVIRONMENT

Keeping track of the time implies that a real-time clock be available in the system; a timer of the MM19 PTM (MC6840) is programmed to fulfill this function.

The MM19 ACIA and PIA are used to interface to the system console and line printer, respectively.

The kernel is the one listed in Appendix A except that PGSTRT is defined as the application program start address $D000; it occupies the EROM sockets U27 and U28.

The MM19 2K-byte RAM starting at $E000 is used to hold the program variables and stacks (data section).

The user code firmware is located in two 2K-byte EROM's located at $D000 (MM19 sockets U29 and U30).

The Runtime package resides externally on a M68MM04 micromodule based at $8000.

3.3. IMPLEMENTATION STEPS

The following steps are taken once the BASICM application program has been edited:

1/. The source program (SAMPLE) is compiled with the following command line:

```
=BASICM SAMPLE;OL=#LP,R=$8000,D=$E000,P=$D000,S
```

The listing shown in appendix B is then obtained on the development system line printer, and the user code is saved in the object file SAMPLE.LO.

As is shown in the previous command line, option "S" was selected for this particular program; this is because the program uses the IRQ interrupt: if option "S" was not specified, the "check break" function would be performed prior to executing each statement, and those of the IRQ service routine in particular; as the "check break" function basically checks the ACIA for the CTRL-P code in its receiver or for a framing error (see kernel), it would then empty the ACIA receiver, thus causing the characters expected by an INPUT statement to be lost.

2/. A single load module is constructed by invoking the BLOAD utility, that comprises the user code (SAMPLE.LO) and the part of the runtime package necessary to this application (Runtime disk I/O are not used).

```
=BLOAD SAMPLE;ORL=$E7FC,-D
```

Note that the DSCT LINK (see paragraph 1.6) is specified at $E7FC, i.e, in the MM19 2K-byte RAM block.

The load module is constructed under the name SAMPLE.CM, and the following map is produced by BLOAD:

```
SAMPLE  .LO:0  D000 - DFB0  (see note (1) )
RUN-TIME       8000 - B10C  (see note (2) )
DSCT LINK      E7FC - E7FD  (see note (3) )
DATA SECTION   E000 - E563  (see note (3) )

LOAD BOUNDS    8000 - DFB0
```

note (1) : will be firmware in MM19 sockets U29, U30.
note (2) : will be firmware on Micromodule M68MM04.
note (3) : in MM19 2K-byte RAM block.

The user just has then to burn into EROM´s the relevant sections of the file SAMPLE.CM whose boundaries are indicated in the map.

The BLOAD command above could have included the kernel as load module; for this application however, the kernel was PROM-programmed from a separate file.

3.4. RUNNING THE PROGRAM

The following shows a sample run of the application program:

```
Results to printer ? N
Random or Input ( R/I ) ? R

given numbers :   6  50   4   2   9  10
Target number to find : 380

                    I made it in  4.10 seconds !

     4  x   6 =   24
    24  +   9 =   33
    33  +  10 =  330
   330  +  50 =  380

                    Quod Erat Demonstrandum.

Results to printer ? N
Random or Input ( R/I ) ? I
Enter the 6 numbers (1..10, 25, 50, 75, 100) : ? 1 2 3 5 7 9
Target number to find ( 101 - 999 ) ? 101

given numbers :   1   2   3   5   7   9
Target number to find : 101

                    I made it in  0.90 seconds !

     7  +   3 =   10
    10  x   5 =   50
    50  x   2 =  100
   100  +   1 =  101

                    Quod Erat Demonstrandum.
```

```
Results to printer ? N
Random or Input ( R/I ) ? R

given numbers :     8 100  8  50  50  9
Target number to find : 580

                    I can't find the solution
```

A P P E N D I X I

M I N I M U M K E R N E L F O R M M 1 9

```
                            NAM     MM19BK
                            OPT     ABS,NOW,LLEN=120
                            TTL     -- BASIC-M MINIMUM KERNEL FOR MM19


                  * MM19 PERIPHERAL DEVICES

EC14     A ACIA     EQU     $EC14      ACIA IS USED FOR MASTER CONSOLE COMMUNICATIONS
EC10     A PIA      EQU     $EC10      PIA IS USED AS CENTRONICS COMPATIBLE PRINTER INTERFACE
EC18     A PTM      EQU     $EC18      NOT USED


                  * SPECIAL CHARACTERS

0004     A EOT      EQU     $04        CTL-D -- END OF TRANSMISSION CHARACTER
000A     A LF       EQU     $0A        LINE FEED
000D     A CR       EQU     $0D        CARRIAGE RETURN
0010     A DLE      EQU     $10        CTL-P -- ABORT CHARACTER
0017     A ETB      EQU     $17        CTL-W -- PAUSE CHARACTER
0020     A SPACE    EQU     $20        SPACE


                  * ACIA CONFIGURATION PARAMETERS

0003     A ACRES    EQU     %11        ACIA MASTER RESET
0000     A ACRINT   EQU     %0         RECEIVE INTERRUPT DISABLED
0000     A ACRTST   EQU     %00        ~RTS LOW, TRANSMIT INTERRUPT DISABLED
0002     A ACPAR    EQU     %010       7 DATA BITS, EVEN PARITY, 1 STOP BIT.
0001     A ACCLK    EQU     %01        DIVIDE CLOCK BY 16 MODE
0009     A ACCTL    EQU     ACRINT!<7+ACRTST!<5+ACPAR!<2+ACCLK ACIA CONTROL WORD


                  * ACIA STATUS BITS

0001     A RDRF     EQU     %1         RECEIVE DATA REGISTER FULL
0002     A TDRE     EQU     %10        TRANSMIT DATA REGISTER EMPTY
0010     A FE       EQU     %10000     FRAMING ERROR


                  * PIA CONFIGURATION PARAMETERS

00FF     A DDRA     EQU     %11111111 ALL LINES A SIDE IN OUTPUT
0000     A DDRB     EQU     %00000000 ALL LINES B SIDE IN INPUT
0000     A IRQA1M   EQU     %0         IRQA1 MASKED
0000     A EDGEA1   EQU     %0         CA1 ACTIVE ON HIGH TO LOW TRANSITION
0001     A CA2DIR   EQU     %1         CA2 IS AN OUTPUT
0003     A CA2      EQU     %11        CA2 ALWAYS HIGH
0000     A IRQB1M   EQU     %0         IRQB1 MASKED
0000     A EDGEB1   EQU     %0         CB1 ACTIVE ON HIGH TO LOW TRANSITION
0001     A CB2DIR   EQU     %1         CB2 IS AN OUTPUT
0003     A CB2      EQU     %11        CB2 ALWAYS HIGH
0038     A PIACTL   EQU     CA2DIR!<5+CA2!<3+EDGEA1!<1+IRQA1M PIA SIDE A CONTROL BYTE
0038     A PIBCTL   EQU     CB2DIR!<5+CB2!<3+EDGEB1!<1+IRQB1M PIA SIDE B CONTROL BYTE
```

```
          0004     A DRSEL   EQU     %100      SELECT DATA REGISTER BIT

                     * DISK ROM EQUATES

          E800     A OSLOAD EQU      $E800     LOAD OS
          E822     A FDINIT EQU      $E822     INITIALIZE DISK INTERFACE
          E875     A RESTOR EQU      $E875     RESTORE DRIVE
          E887     A CLOCK   EQU     $E887     COMPUTE FREQUENCY PARAMETER
                     *
          0000     A CURDRV EQU      $0000     CURRENT DRIVE NUMBER STORAGE

                     * BASIC-M PROGRAM EXECUTION START ADDRESS

          0000     A PGSTRT EQU      $0000     DEFAULT START ADDRESS ( LOAD XDOS )

                             TTL     -- RAM STORAGE
```

```
                           *
                           * USE TOP OF MM19 ONBOARD RAM FOR MONITOR USAGE
                           *
          E7FF     A RAMTOP EQU    $E7FF     HIGHEST RAM ADDRESS
                           *
          E72E     A ATOP   EQU    $E72E     2ND LEVEL INTERRUPT VECTOR TABLE ADDRESS STORAGE
          E7FC     A RAMAD  EQU    $E7FC     DSCT LINK ADDRESS
          E714     A AECHO  EQU    $E714     INPUT CHARACTER ECHO FLAG ( 0 => ECHO )
          E703     A STACK  EQU    $E703     STACK ADDRESS AT RESET TIME


                            TTL    --   JUMP TABLE AND PROGRAM START ADDRESS
```

```
00078A F000                        ORG     $F000
00079A F000        0000    A APSTRT FDB     PGSTRT    BASIC-M PROGRAM START ADDRESS
00080A F002        D000    A ARSTRT FDB     $D000     FATAL ERROR RECOVERY ENTRY POINT ADDRESS
00081A F004        E7FC    A ARAMAD FDB     RAMAD     ADDRESS OF DSCT LINK


00083A F006 17     003F F048        LBSR    UNIMP     CBCDHX UNIMPLEMENTED
00084A F009 16     0065 F071        LBRA    CHEXL     CONVERT MS BCD TO ASCII HEX
00085A F00C 16     0066 F075        LBRA    CHEXR     CONVERT LS BCD TO ASCII HEX
00086A F00F 17     0036 F048        LBSR    UNIMP     INADDR UNIMPLEMENTED
00087A F012 17     0033 F048        LBSR    UNIMP     INCH UNIMPLEMENTED
00088A F015 16     0193 F1AB        LBRA    INCHNP    INPUT CHARACTER, STRIP PARITY
00089A F018 16     0182 F19D XOUTCH LBRA    OUTCH     OUTPUT CHARACTER
00090A F01B 16     0041 F05F        LBRA    OUT2HS    DISPLAY HEX 2 DIGITS
00091A F01E 16     003C F05D        LBRA    OUT4HS    DISPLAY HEX 4 DIGITS
00092A F021 16     0173 F197 XPCRLF LBRA    PCRLF     DISPLAY CARRIAGE-RETURN, LINE-FEED
00093A F024 16     0163 F18A XPDATA LBRA    PDATA     DISPLAY CR, LF, STRING
00094A F027 16     0162 F18C        LBRA    PDATA1    DISPLAY STRING
00095A F02A 16     0034 F061        LBRA    PSPACE    DISPLAY SPACE
00096A F02D 16     0122 F152 XEXIT  LBRA    EXIT      END OF BASIC-M PROGRAM EXECUTION
00097A F030 17     0015 F048        LBSR    UNIMP     XLDA UNIMPLEMENTED
00098A F033 17     0012 F048        LBSR    UNIMP     XSTA UNIMPLEMENTED
00099A F036 17     000F F048        LBSR    UNIMP     XTOGL UNIMPLEMENTED
00100A F039 16     01A4 F1E0        LBRA    ZAPBRK    NO BREAKPOINT, DUMMY ENTRY POINT
00101A F03C 17     0009 F048        LBSR    UNIMP     SAVREC UNIMPLEMENTED
00102A F03F 17     0006 F048        LBSR    UNIMP     GETREG UNIMPLEMENTED
00103A F042 16     01A7 F1EC        LBRA    LIST      OUTPUT CHARACTER TO PRINTER
00104A F045 16     017F F1C7        LBRA    CKBRK     CHECK BREAK CONDITION


00106                               TTL     -- ILLEGAL ENTRY TO MONITOR --
```

```
00108A F048 30   8C 35      UNIMP  LEAX   <ILLENT,PCR
00109A F04B 8D   D7    F024        BSR    XPDATA    DISPLAY "CALL TO UNIMPLEMENTED ROUTINE $"
00110A F04D 35   10       A        PULS   X         COMPUTE ROUTINE ADDRESS
00111A F04F 30   1D       A        LEAX   -3,X
00112A F051 34   10       A        PSHS   X
00113A F053 30   E4       A        LEAX   0,S       DISPLAY ROUTINE ADDRESS
00114A F055 8D   06    F05D        BSR    OUT4HS
00115A F057 8D   C8    F021        BSR    XPCRLF    NEW LINE
00116A F059 32   62       A        LEAS   2,S       DROP ROUTINE ADDRESS
00117A F05B 20   D0    F02D        BRA    XEXIT     ABORT


00119                              TTL    -- NUMBER CONVERSION AND DISPLAY NUMBER ROUTINES --
```

```
00121A F05D 8D   06   F065 OUT4HS BSR    OUT2H    DISPLAY MS BYTE


00123A F05F 8D   04   F065 OUT2HS BSR    OUT2H    DISPLAY BYTE


00125A F061 86   20      A PSPACE LDA    #SPACE
00126A F063 20   B3   F018 YOUTCH BRA    XOUTCH   GO DISPLAY SPACE


00128A F065 A6   84      A OUT2H  LDA    0,X      DISPLAY MS
00129A F067 8D   08   F071        BSR    CHEXL
00130A F069 8D   F8   F063        BSR    YOUTCH
00131A F06B A6   80      A        LDA    0,X+     DISPLAY LS
00132A F06D 8D   06   F075        BSR    CHEXR
00133A F06F 20   F2   F063        BRA    YOUTCH   AND EXIT


00135A F071 44              CHEXL LSRA            MOVE BITS 7-4 TO 3-0
00136A F072 44                    LSRA
00137A F073 44                    LSRA
00138A F074 44                    LSRA


00140A F075 84   0F      A CHEXR  ANDA   #$F      CONVERT LS BCD TO ASCII HEX
00141A F077 8B   30      A        ADDA   #'0
00142A F079 81   39      A        CMPA   #'9
00143A F07B 23   02   F07F        BLS    RTN      0-9
00144A F07D 8B   07      A        ADDA   #7       A-F
00145A F07F 39              RTN   RTS             DONE, EXIT


00147                       *   MESSAGE FOR ERROR ROUTINE
00148                       *
00149A F080      43      A ILLENT FCC    /CALL TO UNIMPLEMENTED ROUTINE $/
00150A F09F      04      A        FCB    EOT


00152                             TTL    -- RESTART RESPONSE ROUTINE -- PERIPHERAL INITIALIZATION
```

```
00154A F0A0 10CE E703     A RES     LDS    #STACK     FIRST THING TO DO IN CASE OF INTERRUPT
00155A F0A4 1A   50       A         ORCC   #$50       MASK IRQ & FIRQ
00156A F0A6 4F                      CLRA              SET DIRECT PAGE = 0
00157A F0A7 1F   8B       A         TFR    A,DP
00158A F0A9 5F                      CLRB              INITIALIZE 2ND LEVEL INTERRUPT VECTOR TABLE ADDRESS
00159A F0AA ED   9D 0F40            STD    [ATOPA,PCR]
00160A F0AE 43                      COMA
00161A F0AF ED   9D FF51            STD    [ARAMAD,PCR] NO BASIC-M PROGRAM IN EXECUTION YET
00162A F0B3 1F   43       A         TFR    S,U        INITIALIZE USER STACK
00163                      *
00164                      * MASTER DEVICE INITIALIZATION
00165                      *
00166A F0B5 86   03       A         LDA    #ACRES     RESET ACIA
00167A F0B7 B7   EC14     A         STA    ACIA
00168A F0BA 86   09       A         LDA    #ACCTL     INITIALIZE IT
00169A F0BC B7   EC14     A         STA    ACIA
00170A F0BF 7F   E714     A         CLR    >AECHO     FORCE ECHO
00171                      *
00172                      * PRINTER INTERFACE INITIALIZATION
00173                      *
00174A F0C2 7F   EC11     A         CLR    PIA+1      SELECT DDRA
00175A F0C5 7F   EC13     A         CLR    PIA+3      SELECT DDRB
00176A F0C8 CC   FF3C     A         LDD    #DDRA!<8+PIACTL+DRSEL
00177A F0CB FD   EC10     A         STD    PIA        INITIALIZE PIA A SIDE
00178A F0CE CC   003C     A         LDD    #DDRB!<8+PIBCTL+DRSEL
00179A F0D1 FD   EC12     A         STD    PIA+2      INITIALIZE PIA B SIDE
00180                      *
00181                      * GO EXECUTE BASIC-M PROGRAM OR LOAD OS
00182                      *
00183A F0D4 AE   8D FF28            LDX    APSTRT,PCR FETCH PROGRAM START ADDRESS
00184A F0D8 26   25   F0FF          BNE    GEXEC      ADDRESS OK?, GO TO IT
00185A F0DA 6F   E2       A         CLR    0,-S       INITIALIZE MESSAGE FLAG
00186A F0DC BD   E822     A         JSR    >FDINIT    ADDRESS IS ZERO, LOAD XDOS
00187A F0DF 7F   0000     A DKWAIT  CLR    >CURDRV    RESTORE DRIVE 0
00188A F0E2 BD   E887     A         JSR    >CLOCK     COMPUTE TIMING PARAMETER
00189A F0E5 25   05   F0EC          BCS    DKNRDY     DISK NOT READY
00190A F0E7 BD   E875     A         JSR    >RESTOR
00191A F0EA 24   0E   F0FA          BCC    GOLOAD     DISK OK, GO LOAD OS
00192A F0EC 6D   E4       A DKNRDY  TST    0,S        MESSAGE PRINTED ?
00193A F0EE 26   EF   F0DF          BNE    DKWAIT     YES, TRY AGAIN
00194A F0F0 30   8C 0E              LEAX   <DKMSG,PCR
00195A F0F3 17   0094 F18A          LBSR   PDATA
00196A F0F6 6C   E4       A         INC    0,S        SET MESSAGE DISPLAYED FLAG
00197A F0F8 20   E5   F0DF          BRA    DKWAIT     GO TRY AGAIN
00198A F0FA 32   61       A GOLOAD  LEAS   1,S        DROP FLAG
00199A F0FC 7E   E800     A         JMP    >OSLOAD    GO LOAD OS
00200                      *
00201A F0FF 6E   84       A GEXEC   JMP    0,X        GO EXECUTE USER PROGRAM
00202                      *
00203A F101      0D       A DKMSG   FCB    CR,LF
00204A F103      49       A         FCC    /INSERT DISK IN DRIVE 0/
00205A F119      0D       A         FCB    CR,LF,EOT


00207                               TTL    -- INTERRUPT HANDLER
```

```
                                 *
                                 * GO EXECUTE SECOND LEVEL INTERRUPT ROUTINE IF ANY
                                 *
                                 * IF ADDRESS FOUND IS ZERO, ABORT WITH "NO VECTOR" MESSAGE
                                 *


                                 * NMI RESPONSE ROUTINE

     A F11C 8D   10   F12E NMI     BSR    INTER      CALL INTERRUPT HANDLER
     A F11E      FD      A         FCB    -3         TOP OF TABLE OFFSET TO NMI VECTOR


                                 * SWI RESPONSE ROUTINE

     A F11F 8D   0D   F12E SWI     BSR    INTER      CALL INTERRUPT HANDLER
     A F121      FB      A         FCB    -5         TOP OF TABLE OFFSET TO SWI VECTOR


                                 * IRQ RESPONSE ROUTINE

     A F122 8D   0A   F12E IRQ     BSR    INTER      CALL INTERRUPT HANDLER
     A F124      F9      A         FCB    -7         TOP OF TABLE OFFSET TO IRQ VECTOR


                                 * FIRQ RESPONSE ROUTINE

     A F125 8D   07   F12E FIRQ    BSR    INTER      CALL INTERRUPT HANDLER
     A F127      F7      A         FCB    -9         TOP OF TABLE OFFSET TO FIRQ VECTOR


                                 * SWI2 RESPONSE ROUTINE

     A F128 8D   04   F12E SWI2    BSR    INTER      CALL INTERRUPT HANDLER
     A F12A      F5      A         FCB    -11        TOP OF TABLE OFFSET TO SWI2 VECTOR


                                 * SWI3 RESPONSE ROUTINE

     A F12B 8D   01   F12E SWI3    BSR    INTER      CALL INTERRUPT HANDLER
     A F12D      F3      A         FCB    -13        TOP OF TABLE OFFSET TO SWI3 VECTOR


                                 * COMMON INTERRUPT HANDLER

     A F12E 34   15      A INTER   PSHS   X,B,CC     SAVE REGISTERS
     A F130 AE   9D 0EBA           LDX    [ATOPA,PCR] FETCH 2ND LEVEL INTERRUPT VECTOR TABLE ADDRESS
     A F134 27   17   F14D         BEQ    NOVEC      NOT SPECIFIED, ERROR
     A F136 E6   F8 04   A         LDB    [4,S]      FETCH TOP OF TABLE OFFSET TO CORRECT VECTOR
     A F139 AE   85      A         LDX    B,X        FETCH VECTOR
     A F13B 27   10   F14D         BEQ    NOVEC      NOT SPECIFIED, ERROR
```

```
00253A F13D AF    64      A          STX    4,S       REPLACE RETURN ADDRESS WITH INTERRUPT ROUTINE ADDRESS
00254A F13F 35    95      A          PULS   CC,B,X,PC RESTORE REGISTERS AND GO TO INTERRUPT ROUTINE
00255                      *
00256A F141       4E      A NVECTM   FCC    /NO VECTOR/
00257A F14A       0D      A          FCB    CR,LF,EOT
00258                      *
00259A F14D 30    8C F1     NOVEC    LEAX   NVECTM,PCR OUTPUT ERROR MESSAGE
00260A F150 8D    38   F18A          BSR    PDATA     AND ABORT


00262                                TTL    -- EXECUTION TERMINATE ROUTINE
```

```
                           *
                           * EXIT FROM BASIC-M PROGRAM
                           *
                           * ENTRY CONDITIONS : NONE
                           *
                           * THIS ROUTINE NEVER RETURNS.
                           *
A F152 1A   50       A EXIT     ORCC   #$50      INHIBIT INTERRUPTS
A F154 10CE E703     A          LDS    #STACK    REINITIALIZE STACK IN CASE OF ENTRY BY CRASH
A F158 EC   9D FEA8             LDD    [ARAMAD,PCR] EXIT FROM BASICM ?
A F15C 27   07   F165           BEQ    RECOV     YES, GO EXECUTE ERROR RECOVERY PROGRAM
A F15E 30   8C 08               LEAX   <EXITM,PCR DISPLAY MESSAGE
A F161 8D   27   F18A           BSR    PDATA
A F163 20   FE   F163           BRA    *         WAIT FOR EVER
A F165 6E   9D FE99    RECOV    JMP    [ARSTRT,PCR] GO EXECUTE ERROR RECOVERY PROGRAM
                           *
A F169      50       A EXITM    FCC    /PRESS RESET TO RESTART PROGRAM/
A F187      0D       A          FCB    CR,LF,EOT


                                TTL    -- INPUT/OUTPUT ROUTINES
```

```
                            *
                            * DISPLAY CR, LF, ASCII STRING TO MASTER DEVICE
                            *
                            * ENTRY CONDITIONS : X = ADDRESS OF ASCII STRING TO DISPLAY
                            *                      STRING IS TERMINATED BY AN EOT ($04) CHARACTER.
                            *
                            * EXIT CONDITIONS  : X = ADDRESS OF BYTE FOLLOWING THE EOT.
                            *                    A IS DESTROYED.
                            *                    CC IS MEANINGLESS.
                            *                    OTHER REGISTERS ARE PRESERVED
                            *
                            * STACK REQUIRMENTS : 8 BYTES.
                            *
A F18A 8D   0B   F197 PDATA  BSR   PCRLF    OUTPUT CARRIAGE-RETURN, LINE-FEED
A F18C A6   80      A PDATA1 LDA   0,X+     FETCH STRING CHARACTER
A F18E 81   04      A        CMPA  #EOT     END OF STRING ?
A F190 27   04   F196        BEQ   PDATA2   YES, EXIT
A F192 8D   09   F19D        BSR   OUTCH    NO, DISPLAY CHARACTER
A F194 20   F6   F18C        BRA   PDATA1   GO FETCH NEXT CHARACTER
A F196 39             PDATA2 RTS            DONE, EXIT


                            *
                            * DISPLAY CR, LF ON MASTER DEVICE
                            *
                            * ENTRY CONDITIONS : NONE
                            *
                            * EXIT CONDITIONS  : A IS DESTROYED
                            *                    CC IS MEANINGLESS
                            *                    OTHER REGISTERS ARE PRESERVED
                            *
                            * STACK REQUIRMENTS : 5 BYTES
                            *
A F197 86   0D      A PCRLF  LDA   #CR      OUTPUT CARRIAGE RETURN
A F199 8D   02   F19D        BSR   OUTCH
A F19B 86   0A      A        LDA   #LF      OUTPUT LINE FEED, FALL IN OUTCH ROUTINE


                            *
                            * DISPLAY CHARACTER ON MASTER DEVICE
                            *
                            * ENTRY CONDITIONS : A = CHARACTER TO DISPLAY
                            *
                            * EXIT CONDITIONS  : CC IS MEANINGLESS
                            *                    OTHER REGISTERS ARE PRESERVED
                            *
                            * STACK REQUIRMENTS : 3 BYTES
                            *
A F19D 34   04      A OUTCH  PSHS  B        SAVE B REGISTER
A F19F C6   02      A        LDB   #TDRE    WAIT FOR ACIA READY
A F1A1 F5   EC14    A OUTCH2 BITB  ACIA
A F1A4 27   FB   F1A1        BEQ   OUTCH2   NOT READY, WAIT
A F1A6 B7   EC15    A        STA   ACIA+1   ACIA READY, SEND CHARACTER
A F1A9 35   84      A        PULS  B,PC     RESTORE B REGISTER AND EXIT
```

```
                             *
                             * INPUT A CHARACTER FROM MASTER DEVICE, STRIP PARITY, ECHO CHARACTER
                             *
                             * ENTRY CONDITIONS : NONE
                             *
                             * EXIT CONDITIONS  : A CONTAINS THE INPUT CHARACTER
                             *                    CC IS MEANINGLESS
                             *                    OTHER REGISTERS ARE PRESERVED
                             *
                             * STACK REQUIRMENTS : 3 BYTES
                             *
A F1AB 86   01        A INCHNP LDA   #RDRF    TEST ACIA STATUS
A F1AD B5   EC14      A INCH1  BITA  ACIA     WAIT FOR ACIA READY
A F1B0 27   FB   F1AD          BEQ   INCH1    NOT READY
A F1B2 B6   EC15      A        LDA   ACIA+1   FETCH INPUT CHARACTER
A F1B5 84   7F        A        ANDA  #%01111111 STRIP PARITY
A F1B7 7D   E714      A        TST   >AECHO   ECHO CHARACTER ?
A F1BA 27   E1   F19D          BEQ   OUTCH    YES, GO DO IT
A F1BC 7F   E714      A        CLR   >AECHO   RESET FLAG
A F1BF 39                      RTS            AND EXIT


                             *
                             * TEST BREAK CONDITION
                             *
                             * ENTRY CONDITIONS : NONE
                             *
                             * EXIT CONDITIONS  : C = 0  OK, CONTINUE
                             *                      = 1  A BREAK CONDITION HAS OCCURED
                             *                    REMAINDER OF CC IS MEANINGLESS
                             *                    A IS DESTROYED
                             *                    OTHER REGISTERS ARE PRESERVED
                             *
                             * STACK REQUIRMENTS : 2 BYTES
                             *
                             *
                             * BREAK CONDITION IS EITHER A CONTROL-P OR A BREAK KEY-IN
                             *
                             * IF CONTROL-W HAS BEEN TYPED, THE ROUTINE WAITS FOR ANOTHER CHARACTER
                             * BEFORE RETURNING TO CALLING PROGRAM.
                             *
                             * ROUTINE ENTRY POINT IS AT CKBRK
                             *
A F1C0 86   11        A BRK3   LDA   #FE+RDRF TEST FRAMING ERROR OR CHARACTER RECEIVED
A F1C2 B5   EC14      A BRK4   BITA  ACIA     WAIT UNTIL ONE OR BOTH DETECTED
A F1C5 27   FB   F1C2          BEQ   BRK4
                             *
A F1C7 B6   EC14      A CKBRK  LDA   ACIA     FETCH STATUS
A F1CA 85   10        A        BITA  #FE      FRAMING ERROR ?
A F1CC 26   13   F1E1          BNE   BRK1     YES, BREAK KEY HAS BEEN HIT
A F1CE 85   01        A        BITA  #RDRF    CHARACTER READY ?
A F1D0 27   0D   F1DF          BEQ   BRK2     NO, EXIT
A F1D2 B6   EC15      A        LDA   ACIA+1   YES, FETCH CHARACTER
A F1D5 84   7F        A        ANDA  #$7F     STRIP PARITY
A F1D7 81   10        A        CMPA  #DLE     CONTROL-P ?
A F1D9 27   06   F1E1          BEQ   BRK1     YES, GO EXIT WITH CARRY SET
A F1DB 81   17        A        CMPA  #ETB     CONTROL-W ?
```

```
A F1DD 27   E1   F1C0          BEQ   BRK3      YES, GO WAIT FOR ANOTHER CHARACTER
A F1DF 4F                BRK2  CLRA            NO BREAK CONDITION, CLEAR CARRY
A F1E0 39                ZAPBRK RTS            AND EXIT
                         *
A F1E1 B6   EC15      A BRK1   LDA   ACIA+1    A BREAK CONDITION HAS OCCURED, RESET FRAMING ERROR
A F1E4 B6   EC15      A        LDA   ACIA+1    AND POSSIBLE OVERRUN
A F1E7 B6   EC15      A        LDA   ACIA+1
A F1EA 43                      COMA            SET CARRY
A F1EB 39                      RTS             EXIT


                         *
                         * OUTPUT CHARACTER TO PRINTER
                         *
                         * ENTRY CONDITIONS : A = CHARACTER TO OUTPUT
                         *
                         * EXIT CONDITIONS  : C = 0 => OUTPUT WAS SUCCESSFUL
                         *                        = 1 => OUTPUT FAILED
                         *                    REMAINDER OF CC IS MEANINGLESS
                         *                    OTHER REGISTERS ARE PRESERVED
                         *
                         * STACK REQUIRMENTS : 3 BYTES
                         *
A F1EC B7   EC10      A LIST   STA   PIA       PUT CHARACTER ON PRINTER BUS
A F1EF B6   EC10      A        LDA   PIA       CLEAR PREVIOUS STATUS
A F1F2 86   34        A        LDA   #$34      STROBE DATA
A F1F4 B7   EC11      A        STA   PIA+1
A F1F7 86   3C        A        LDA   #$3C
A F1F9 B7   EC11      A        STA   PIA+1
A F1FC 43                      COMA            SET ERROR STATUS
A F1FD B6   EC12      A LIST1  LDA   PIA+2     FETCH PRINTER STATUS
A F200 84   03        A        ANDA  #3
A F202 4A                      DECA            MUST BE ONLINE
A F203 26   06   F20B          BNE   LIST2     ERROR, OUTPUT FAILED
A F205 7D   EC11      A        TST   PIA+1     WAIT UNTIL CHARACTER OUT
A F208 2A   F3   F1FD          BPL   LIST1     NOT OUT, LOOP
A F20A 4F                      CLRA            OK, EXIT WITH CARRY CLEAR
A F20B B6   EC10      A LIST2  LDA   PIA       RESTORE A REGISTER, CLEAR STATUS
A F20E 39                      RTS             AND EXIT


                               TTL   -- HARDWARE INTERRUPT VECTORS
```

```
                              *
                              * ASSIGN TO TOP OF ADDRESSABLE MEMORY
                              *
A FFEE                               ORG    $FFEE
A FFEE          E72E      A ATOPA    FDB    ATOP      ADDRESS OF 2ND LEVEL INTERRUPT VECTOR TABLE ADDRESS
                              *
A FFF0          F14D      A          FDB    NOVEC     RESERVED, ERROR
A FFF2          F12B      A          FDB    SWI3      SOFTWARE INTERRUPT 3 VECTOR
A FFF4          F128      A          FDB    SWI2      SOFTWARE INTERRUPT 2 VECTOR
A FFF6          F125      A          FDB    FIRQ      FAST INTERRUPT REQUEST VECTOR
A FFF8          F122      A          FDB    IRQ       INTERRUPT REQUEST VECTOR
A FFFA          F11F      A          FDB    SWI       SOFTWARE INTERRUPT 1 VECTOR
A FFFC          F11C      A          FDB    NMI       NON MASKABLE INTERRUPT VECTOR
A FFFE          F0A0      A          FDB    RES       RESTART VECTOR


                F0A0      A          END    RES       END OF MM19BK. P.MONNERAT, OCT. 15, 1980.
TOTAL ERRORS 00000--00000


ACCLK  0001 ACCTL  0009 ACIA   EC14 ACPAR  0002 ACRES  0003 ACRINT 0000 ACRTST 0000 AECHO  E714 APSTRT F000
ARAMAD F004 ARSTRT F002 ATOP   E72E ATOPA  FFEE BRK1   F1E1 BRK2   F1DF BRK3   F1C0 BRK4   F1C2 CA2    0003
CA2DIR 0001 CB2    0003 CB2DIR 0001 CHEXL  F071 CHEXR  F075 CKBRK  F1C7 CLOCK  E887 CR     000D CURDRV 0000
DDRA   00FF DDRB   0000 DKMSG  F101 DKNRDY F0EC DKWAIT F0DF DLE    0010 DRSEL  0004 EDGEA1 0000 EDGEB1 0000
EOT    0004 ETB    0017 EXIT   F152 EXITM  F169 FDINIT E822 FE     0010 FIRQ   F125 GEXEC  F0FF GOLOAD F0FA
ILLENT F080 INCH1  F1AD INCHNP F1AB INTER  F12E IRQ    F122 IRQA1M 0000 IRQB1M 0000 LF     000A LIST   F1EC
LIST1  F1FD LIST2  F20B NMI    F11C NOVEC  F14D NVECTM F141 OSLOAD E800 OUT2H  F065 OUT2HS F05F OUT4HS F05D
OUTCH  F19D OUTCH2 F1A1 PCRLF  F197 PDATA  F18A PDATA1 F18C PDATA2 F196 PGSTRT 0000 PIA    EC10 PIACTL 0038
PIBCTL 0038 PSPACE F061 PTM    EC18 RAMAD  E7FC RAMTOP E7FF RDRF   0001 RECOV  F165 RES    F0A0 RESTOR E875
RTN    F07F SPACE  0020 STACK  E703 SWI    F11F SWI2   F128 SWI3   F12B TDRE   0002 UNIMP  F048 XEXIT  F02D
XOUTCH F018 XPCRLF F021 XPDATA F024 YOUTCH F063 ZAPBRK F1E0
```

A P P E N D I X J

A P P L I C A T I O N P R O G R A M L I S T I N G

```
006C   00010  REM   *****  SAMPLE PROGRAM FOR MM19   *****
006C   00020  REM
006C   00030  INTEGER Time,Dummy
006C   00040  BYTE Tswch,Tmout
006C   00050  BYTE PTM_cr13 ADDRESS $EC18
006C   00060  BYTE PTM_cr2 ADDRESS $EC19
006C   00070  BYTE PTM_stat ADDRESS PTM_cr2
006C   00080  INTEGER PTM_t1 ADDR $EC1A
006C   00090  INTEGER PTM_t2 ADDR $EC1C
006C   00100  INTEGER PTM_t3 ADDR $EC1E
006C   00110  INTEGER Prnt,LU,X,I,J,M,P,CI,CA,CN,CT,Zero,One,Hundred
006C   00120  INTEGER A(12),N(7),II(12),T(7),Numbers(14)
006C   00130  REM
006C   00140  DATA $1,$2,$3,$4,$5,$6,$7,$8,$9,$A,$19,$32,$4B,$64
006C   00150  DATA $0,$1,$64
006C   00160  REM
006C   00170  DEF Odd= IAND(M,One)
008F   00180  REM
008F   00190  MAT  READ Numbers
0096   00200  READ Zero,One,Hundred
00AF   00210  REM
00AF   00220  PTM_cr2=One             \ Select CR1
00BE   00230  PTM_cr13=One            \ Clear CR1, preset state
00CD   00240  PTM_cr2=Zero            \ Select CR3
00DC   00250  PTM_cr13=Zero           \ Clear CR3
00EB   00260  PTM_cr2=One             \ Clear CR2, select CR1
00FA   00270  Tswch=Zero
0109   00280  GOSUB 1710              \ Start timer for randomize
010E   00290  INPUT "Results to printer ",R$
0133   00300  I= SUBSTR("YESNO",R$)
015D   00310  Prnt=One
016C   00320  IF I=4 THEN 370
018F   00330  Prnt=$2
01A2   00340  IF I=1 THEN 370
01C5   00350  PRINT "Answer YES or NO, please !!!"
01F1   00360  GOTO 290
01F6   00370  P=Zero
0205   00380  INPUT "Random or Input ( R/I )  ",R$
022F   00390  ON  SUBSTR("RI", LEFT$(R$,1))+One GOTO 400,420,510
0287   00400  PRINT "Invalid answer"
02A5   00410  GOTO 370
02AA   00420  J= RND(PTM_t1/10000)
02D8   00430  FOR I=One TO $6 STEP One
0303   00440  CN=Numbers( FIX(14* RND )+One)
0343   00450  N(I)=CN
035C   00460  IF CN=Hundred THEN P=CN
037F   00470  NEXT I
0382   00480  X= FIX(900* RND )+Hundred
03B8   00490  IF X=P THEN 480
03D1   00500  GOTO 680
03D6   00510  PRINT "Enter the ?6 numbers (1..10, 25, 50, 75, 100) : ";
0415   00520  INPUT N(One),N($2),N($3),N($4),N($5),N($6)
049A   00530  FOR I=One TO $6 STEP One
04C5   00540  CN=N(I)
04DE   00550  IF CN=Hundred THEN P=One
0501   00560  FOR J=One TO $E STEP One
052C   00570  IF CN=Numbers(J) THEN 610
054F   00580  NEXT J
0552   00590  PRINT "Invalid input"
```

```
056F    00600   GOTO 510
0574    00610   NEXT I
0577    00620   P=P+Hundred
058F    00630   PRINT "Target number to find ( ";P;"- 999 ) ";
05CD    00640   INPUT X
05DF    00650   IF X>=P AND X<$03E8 THEN 680
060E    00660   PRINT "Invalid number"
062C    00670   GOTO 630
0631    00680   FOR LU=One TO Prnt STEP One
0658    00690   PRINT #LU  USING "[/3] given numbers : [3]",N(One);
069D    00700   FOR I=$2 TO $6 STEP One
06CC    00710   PRINT #LU  USING " [3]",N(I);
06FD    00720   NEXT I
0700    00730   PRINT #LU
0716    00740   PRINT #LU  USING " Target number to find : [3]",X
0755    00750   NEXT LU
0758    00760   GOSUB 1780              \ Stop timer
075D    00770   Tswch=One               \ Count time
076C    00780   Time=Zero               \ Initialize time accumulator
077B    00790   Tmout=Zero              \ No timeout yet
078A    00800   GOSUB 1710              \ Start timer
078F    00810   A(One)=X
07A8    00820   MAT II= ZER
07B6    00830   MAT T= ZER
07C4    00840   N($7)=Zero
07E1    00850   M=Zero
07F0    00860   M=M+One
0808    00870   P=Odd
0819    00880   CI=II(M)
0832    00890   CI=CI+One
084A    00900   IF CI<$7+P THEN 1020
0870    00910   II(M)=Zero
0889    00920   M=M-One
08A1    00930   IF M=Zero THEN 1200
08BA    00940   P=Odd
08CB    00950   CI=II(M)
08E4    00960   CT=T(CI)
08FD    00970   IF CT=$FFFF THEN 1000
091A    00980   T(CI)=Zero
0933    00990   GOTO 890
0938    01000   CT=One
0947    01010   GOTO 1050
094C    01020   CT=T(CI)
0965    01030   IF CT<>Zero THEN 890
097E    01040   IF CI<>$7 THEN CT=$2-$3*P
09BF    01050   IF Tmout<>Zero THEN 1200
09DB    01060   II(M)=CI
09F4    01070   T(CI)=CT
0A0D    01080   CA=A(M)
0A26    01090   CN=N(CI)
0A3F    01100   IF P=Zero THEN 1140
0A58    01110   J=CA+CT*CN
0A79    01120   IF J=Zero THEN 1250
0A92    01130   GOTO 1170
0A97    01140   J=CA/CN
0AAF    01150   IF J*CN<>CA THEN 980
0AD1    01160   IF J=One THEN 1250
0AEA    01170   IF M=$C THEN 980
0B07    01180   A(M+One)=J
```

```
0B29    01190   GOTO 860
0B2E    01200   GOSUB 1780                          \ Stop timer
0B33    01210   FOR LU=One TO Prnt STEP One
0B5A    01220   PRINT #LU  USING 1540
0B71    01230   NEXT LU
0B74    01240   GOTO 270
0B79    01250   GOSUB 1780                          \ Stop timer
0B7E    01260   FOR LU=One TO Prnt STEP One
0BA5    01270   PRINT #LU  USING 1550,Time/100
0BD8    01280   NEXT LU
0BDB    01290   J=N(II(M))
0BFE    01300   M=M-One
0C16    01310   CI=II(M)
0C2F    01320   CT=T(CI)
0C48    01330   CN=N(CI)
0C61    01340   CA=A(M)
0C7A    01350   IF Odd<>Zero THEN 1410
0C98    01360   IF CN=One THEN 1460
0CB1    01370   FOR LU=One TO Prnt STEP One
0CD8    01380   PRINT #LU  USING 1510,J,CN,CA
0D0E    01390   NEXT LU
0D11    01400   GOTO 1450
0D16    01410   IF CN=Zero THEN 1460
0D2F    01420   FOR LU=One TO Prnt STEP One
0D56    01430   PRINT #LU  USING 1520,J,-CN*CT,CA
0D98    01440   NEXT LU
0D9B    01450   J=CA
0DAA    01460   IF M<>One THEN 1300
0DC3    01470   FOR LU=One TO Prnt STEP One
0DEA    01480   PRINT #LU  USING 1530
0E01    01490   NEXT LU
0E04    01500   GOTO 270
0E09    01510   IMAGE "[9]   x[5]  =[5]"
0E1B    01520   IMAGE "[9]   [C+()5]  =[5]"
0E30    01530   IMAGE "[/2][X22]Quod Erat Demonstrandum.[/3]"
0E58    01540   IMAGE "[/3][X20]I can´t find the solution[/3]"
0E81    01550   IMAGE "[/3][X22]I made it in[3,2] seconds ![/]"
0EAB    01560   REM
0EAB    01570   REM  -----  IRQ RESPONSE ROUTINE  -----
0EAB    01580   REM
0EAB    01590   IF PTM_stat=$81 THEN 1630
0ECB    01600   PTM_cr13=One                       \ IRQ does not come from PTM
0EDA    01610   STOP "BAD IRQ"
0EE6    01620   REM
0EE6    01630   IF Tswch=Zero THEN 1660
0F02    01640   Time=Time+One                      \ 10 MORE MILLISECONDS
0F1A    01650   IF Time>=$1770 THEN Tmout=One \ Timeout after 60 seconds
0F41    01660   Dummy=PTM_t1                       \ Clear IRQ
0F50    01670   RETURN
0F51    01680   REM
0F51    01690   REM  -----  START TIMER ROUTINE  -----
0F51    01700   REM
0F5?1    01710   PTM_t1=$270F                       \ Count 10 ms
0F64    01720   ON IRQ  THEN  GOSUB 1590
0F6F    01730   PTM_cr13=$42                       \ Start timer, enable interrupts
0F82    01740   RETURN
0F83    01750   REM
0F83    01760   REM  -----  STOP TIMER ROUTINE  -----
0F83    01770   REM
```

```
0F83    01780   PTM_crl3=One          \ Inhibit interrupts, hold in preset state
0F92    01790   NEVER  IRQ
0F95    01800   RETURN
0F96    01810   END
```

```
Time....................I......E000.....
Dummy...................I......E002.....
Tswch...................B......E004.....
Tmout...................B......E005.....
PTM_cr13................B......EC18.....
PTM_cr2.................B......EC19.....
PTM_stat................B......EC19.....
PTM_t1..................I......EC1A.....
PTM_t2..................I......EC1C.....
PTM_t3..................I......EC1E.....
Prnt....................I......E006.....
LU......................I......E008.....
X.......................I......E00A.....
I.......................I......E00C.....
J.......................I......E00E.....
M.......................I......E010.....
P.......................I......E012.....
CI......................I......E014.....
CA......................I......E016.....
CN......................I......E018.....
CT......................I......E01A.....
Zero....................I......E01C.....
One.....................I......E01E.....
Hundred.................I......E020.....
A.......................I......E022....1
N.......................I......E03C....1
II......................I......E04C....1
T.......................I......E066....1
Numbers.................I......E076....1
Odd.....................RF.....0071.....
R$......................S......E094.....

DSCT: E000-E563          PSCT: D000-DFB0
RUNTIME BASE : 8000
END OF COMPILATION
```