

# MPF-IP

## BASIC Manual





# **MPF-IP BASIC PROGRAMMIING MANUAL**

## COPYRIGHT

Copyright © 1983 by Multitech Industrial Corp. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Multitech Industrial Corp.

## DISCLAIMER

Multitech Industrial Corp. makes no representations or warranties, either express or implied, with respect to the contents hereof and specifically disclaims any warranties or merchantability or fitness for any particular purpose. Multitech Industrial Corp. software described in this manual is sold or licensed "as is". Should the programs prove defective following their purchase, the buyer (and not Multitech Industrial Corp., its distributor, or its dealer) assumes the entire cost of all necessary servicing, repair, and any incidental or consequential damages resulting from any defect in the software. Further, Multitech Industrial Corp. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Multitech Industrial Corp. to notify any person of such revision or changes.



**Multitech** INDUSTRIAL  
CORP.

OFFICE/ 315 FU HSIN N. RD., TAIPEI, TAIWAN, R.O.C.  
FACTORY/ 1 INDUSTRIAL E. RD., III HSINCHU SCIENCE-BASED  
INDUSTRIAL PARK, HSINCHU, TAIWAN, R.O.C.

# Table of Contents

---

|           |                                                 |      |
|-----------|-------------------------------------------------|------|
| Chapter 1 | Introduction to BASIC.....                      | 1-1  |
| 1.1       | Special Keys.....                               | 1-1  |
| 1.2       | Prompt Characters.....                          | 1-2  |
| 1.3       | Entry into and Exit from the BASIC system.....  | 1-3  |
| 1.4       | Correction of Errors While Inputing a Program.. | 1-4  |
| 1.5       | BASIC Commands and Statements.....              | 1-4  |
| 1.5.1     | Execution Modes.....                            | 1-5  |
| 1.5.2     | Commands.....                                   | 1-6  |
| 1.5.3     | Statements.....                                 | 1-7  |
| 1.5.4     | Correct or Delete a Statement.....              | 1-8  |
| 1.6       | Listing of a Program.....                       | 1-10 |
| 1.7       | Execution of a Program.....                     | 1-11 |
| 1.8       | Deletion of a Whole Program.....                | 1-11 |
| 1.9       | Remark in a Program.....                        | 1-12 |
| 1.10      | Usage of:.....                                  | 1-13 |
| Chapter 2 | Expression.....                                 | 2-1  |
| 2.1       | Constants.....                                  | 2-2  |
| 2.1.1     | Numeric Constants.....                          | 2-2  |
| 2.1.2     | Literal Strings.....                            | 2-4  |
| 2.2       | Variables.....                                  | 2-5  |



|     |                             |     |
|-----|-----------------------------|-----|
| 2.3 | Functions.....              | 2-6 |
| 2.4 | Operators.....              | 2-8 |
| 2.5 | Evaluating Expressions..... | 2-9 |

## Chapter 3 Commands.....3-1

|       |                                 |      |
|-------|---------------------------------|------|
| 3.1   | Execution Commands.....         | 3-2  |
| 3.1.1 | RUN/XEQ/GOTO.....               | 3-2  |
| 3.1.2 | Continue.....                   | 3-5  |
| 3.1.3 | QUIT.....                       | 3-7  |
| 3.2   | Editing Commands.....           | 3-8  |
| 3.2.1 | LIST.....                       | 3-8  |
| 3.2.2 | NEW/NEW*.....                   | 3-10 |
| 3.2.3 | EDIT.....                       | 3-10 |
| 3.3   | Permanent Storage Commands..... | 3-12 |
| 3.3.1 | SAVE.....                       | 3-12 |
| 3.3.2 | LOAD.....                       | 3-13 |
| 3.4   | Auxiliary Commands.....         | 3-14 |
| 3.4.1 | FREE.....                       | 3-15 |
| 3.4.2 | HEX.....                        | 3-16 |

## Chapter 4 General Statements.....4-1

|     |               |     |
|-----|---------------|-----|
| 4.1 | LET.....      | 4-1 |
| 4.2 | END/STOP..... | 4-3 |

|                             |                                          |      |
|-----------------------------|------------------------------------------|------|
| 4.3                         | REM and "!"                              | 4-4  |
| 4.4                         | RANDOMIZE                                | 4-5  |
| 4.5                         | SON (Speed On) and SOFF (Speed Off)      | 4-6  |
| Chapter 5 Control Statement |                                          | 5-1  |
| 5.1                         | Loop                                     | 5-1  |
| 5.1.1                       | FOR                                      | 5-3  |
| 5.1.2                       | NEXT                                     | 5-3  |
| 5.1.3                       | FOR/NEXT LOOP                            | 5-4  |
| 5.1.4                       | Some Examples                            | 5-6  |
| 5.2                         | Conditional Control Transfer             | 5-9  |
| 5.2.1                       | IF...THEN                                | 5-9  |
| 5.2.2                       | More on Loops                            | 5-11 |
| 5.3                         | Unconditional Control Transfer           | 5-13 |
| 5.3.1                       | GOTO                                     | 5-13 |
| 5.3.2                       | Infinite Loop                            | 5-14 |
| 5.4                         | Computer Control Transfer                | 5-16 |
| 5.4.1                       | ON...GOTO                                | 5-16 |
| Chapter 6 Numeric Operation |                                          | 6-1  |
| 6.1                         | The Notation of Numeric Values in Memory | 6-1  |
| 6.2                         | umeric Functions                         | 6-2  |
| 6.2.1                       | ABS                                      | 6-2  |
| 6.2.2                       | ATN                                      | 6-2  |

|                                     |                                          |      |
|-------------------------------------|------------------------------------------|------|
| 6.2.3                               | COS.....                                 | 6-3  |
| 6.2.4                               | EXP.....                                 | 6-4  |
| 6.2.5                               | INT.....                                 | 6-5  |
| 6.2.6                               | LN.....                                  | 6-5  |
| 6.2.7                               | LOG.....                                 | 6-6  |
| 6.2.8                               | RND.....                                 | 6-7  |
| 6.2.9                               | SGN.....                                 | 6-9  |
| 6.2.10                              | SIG.....                                 | 6-9  |
| 6.2.11                              | SQR.....                                 | 6-10 |
| 6.2.12                              | TAN.....                                 | 6-11 |
| <br>Chapter 7 Array.....            |                                          | 7-1  |
| 7.1                                 | DIM.....                                 | 7-2  |
| 7.2                                 | Changing the Dimensionn of an Array..... | 7-4  |
| <br>Chapter 8 String Operation..... |                                          | 8-1  |
| 8.1                                 | String Literals.....                     | 8-1  |
| 8.2                                 | String Variable.....                     | 8-2  |
| 8.3                                 | String Expression.....                   | 8-3  |
| 8.4                                 | Functions for Strings.....               | 8-4  |
| 8.4.1                               | ASCII.....                               | 8-5  |
| 8.4.2                               | CHR\$......                              | 8-7  |
| 8.4.3                               | INSTR.....                               | 8-8  |

|        |                              |      |
|--------|------------------------------|------|
| 8.4.4  | LEFT\$.....                  | 8-10 |
| 8.4.5  | LEN.....                     | 8-11 |
| 8.4.6  | MID\$.....                   | 8-12 |
| 8.4.7  | NUM\$.....                   | 8-13 |
| 8.4.8  | RIGHT\$.....                 | 8-15 |
| 8.4.9  | SPACES\$.....                | 8-16 |
| 8.4.10 | STRING\$.....                | 8-18 |
| 8.4.11 | VAL.....                     | 8-20 |
| 8.5    | Comparing Strings.....       | 8-22 |
| 8.6    | Input/Output of Strings..... | 8-24 |

## Chapter 9 I/O Statement.....9-1

|       |                                          |      |
|-------|------------------------------------------|------|
| 9.1   | Print.....                               | 9-2  |
| 9.1.1 | Output of Numeric Data.....              | 9-4  |
| 9.1.2 | Output of String Data.....               | 9-5  |
| 9.1.3 | The Usage of "," in a PRINT Statement... | 9-6  |
| 9.1.4 | The Usage of ";" in a PRING Statement... | 9-7  |
| 9.1.5 | Omission of ";".....                     | 9-7  |
| 9.1.6 | POS Function.....                        | 9-8  |
| 9.1.7 | TAB Function.....                        | 9-9  |
| 9.2   | INPUT Statement.....                     | 9-10 |
| 9.2.1 | Input of Numeric Data.....               | 9-12 |
| 9.2.2 | Input of String Data.....                | 9-12 |

|       |                        |      |
|-------|------------------------|------|
| 9.3   | DATA/READ/RESTORE..... | 9-13 |
| 9.3.1 | Examples.....          | 9-15 |
| 9.4   | INP Function.....      | 9-16 |
| 9.5   | OUT Statement.....     | 9-16 |

## Chapter 10 Subprogram.....10-1

|        |                                 |      |
|--------|---------------------------------|------|
| 10.1   | Components of a Subprogram..... | 10-2 |
| 10.2   | GOSUB.....                      | 10-3 |
| 10.2.1 | GOSUB.....                      | 10-3 |
| 10.2.2 | ON/GOSUB.....                   | 10-4 |
| 10.2.3 | RETURN.....                     | 10-5 |
| 10.2.4 | Examples.....                   | 10-6 |
| 10.3   | Recursive Subprogram.....       | 10-8 |

## Chapter 11 User-Defined Function.....11-1

|      |                                    |      |
|------|------------------------------------|------|
| 11.1 | DEF Statement.....                 | 11-2 |
| 11.2 | Usage of User-Defied Function..... | 11-4 |

## Chapter 12 Combination with Non-BASIC Program.....12-1

|      |                     |      |
|------|---------------------|------|
| 12.1 | CALL Statement..... | 12-1 |
| 12.2 | POKE/PEEK.....      | 12-4 |

|            |                                     |     |
|------------|-------------------------------------|-----|
| Appendix A | ASCII Characters.....               | A-1 |
| Appendix B | MPF-IP BASIC Statement.....         | B-1 |
| Appendix C | MPF-IP BASIC Commands.....          | C-1 |
| Appendix D | MPF-IP BASIC Built-in Function..... | D-1 |
| Appendix E | MPF-IP BASIC Error Messages.....    | E-1 |
| Appendix F | Fundamental Definitions.....        | F-1 |
| Appendix G | Ways to Save Memory.....            | G-1 |
| Appendix H | Library Constant.....               | H-1 |
| Appendix I | Some Subprogram in the Monitor..... | I-1 |

## PREFACE

This book is written for those who want to program on MPF-IP in BASIC. In fact, there are quite a few versions of BASIC, each one with its own features designed for special considerations required in different fields.

MPF-IP BASIC, like other versions of BASIC, consists of a set of instructions which you combine to create programs which in turn define the tasks you want the computer to do.

In this manual, detailed descriptions are provided on all the statements, commands and functions available in MPF-IP BASIC. For novice programmers this manual will serve well as a tutorial. Readers who have had experience in programming in other versions of BASIC may read through this manual in a short period of time to familiarize themselves with the special features of MPF-IP BASIC.

This manual is divided into 12 chapters together with an informative section of Appendices. It is hoped that you learn and enjoy more and more in programming as you finish each section of the book.

---

# Chapter 1

## Introduction to BASIC

---

In order to communicate with a computer, we have to learn a language that the computer can understand. In this manual, we will show to the readers a computer language called "BASIC" - the abbreviation of "Beginner's All-purpose Symbolic Instruction Code". BASIC was brought into being by John Kemeny and Thomas in the middle of 1960's. Among the numerous computer languages currently in existence, BASIC is most widely used by people because it is easy to learn, and versatile in application.

MPF-IP BASIC has a number of commands in hand to control the execution of programs as well as a number of statements to write programs. It may well be regarded as a dialect of the BASIC. It is designed to execute on the MPF-IP system and inputs programs and data through the keyboard. In this chapter, we will show you how to make the MPF-IP go into the BASIC system and leave it, and how to input or correct BASIC commands and statements.

### 1.1 Special Keys

 :

One has to end each and every command or statement by pressing this key.



<-- :

This is used to cancel the character before the cursor on the same line.

--> :

The MPF-IP Display is a Fluorescent Indicator Panel (FIP) with a length of 20 characters, but the length of the Display buffer reaches 60 characters. With this key one can see those after the 21st character.

**CONTROL** M :

This 2-key combination serves the same purpose as that of .

**CONTROL** C +  :

This can interrupt the execution of a program.

**RESET** :

After MPF-IP System is started, a press on this key at any moment will bring a display of \*\*\*\*\* MPF-I-PLUS \*\*\*\*\* back on the Indicator Panel.

## 1.2 Prompt Characters

MPF-IP system uses a number of prompt characters to keep the users aware of its present state of being.

- <: The presence of this prompt character indicates that MPF-IP is under the control of the Monitor and a press on CONTROL B or CONTROL C will bring it into the BASIC.
- @: This prompt character tells us that MPF-IP is now in the BASIC and we can input any BASIC commands or statements.
- ?: When an INPUT statement is executed in the user's Program, this prompt character will appear on the Indicator Panel and MPF-IP is awaiting input.

## 1.3 Entry into and Exit from the BASIC System

After MPF-IP is turned on and a display of \*\*\*\*\* MPF-I-PLUS \*\*\*\*\* or the fundamental mode <^ is shown, there are two ways to enter into the BASIC language:

- (1) Press **CONTROL** and B simultaneously (for BASIC), MPF-IP will respond with

BASIC-IP, ORG : ^

and now MPF-IP is expecting the user to enter program storage starting address in hexadecimal and press **←**, if the user press **←** without entering the starting address, the starting address will automatically be set to the default value (the default starting address).

MPF-IP will automatically detect the RAM space, if there is only one RAM on the MPF-IP mainboard, the default value is set to F800, if there are two, then it is set to F000, if an EPB-MPF-IP or IOM-MPF-IP is connected, then the default starting address is set to D800.

Once the starting address of the program is determined, the BASIC prompt character "@^" is displayed on the FIP.

- (2) Press **CONTROL** and C simultaneously: (for REBASIC) MPF-IP will enter into BASIC system directly and the prompt character of the BASIC system "@^" will appear on the indicator panel.

The most important difference between BASIC and REBASIC lies in their manipulation on the memory. In BASIC, the contents of the memory will change as a result of the initialization. In REBASIC, however, there is no initialization and all the data originally in the memory are protected from being destroyed.

### Exit from the BASIC language

There are two ways for exiting from the BASIC language.

- (1) Press the **RESET** key

Whenever the **RESET** key is pressed, the MPF-IP is initialized to its start state, i.e., a display of \*\*\*\*\* MPF-I-PLUS \*\*\*\*\* will be seen on the indicator panel.

- (2) When the BASIC prompt character is displayed on the Indicator Panel, key in QUIT and **←**. Then the control will be returned to the monitor program and the prompt character of the monitor program <^ will be seen on the display.

After you have exited from the BASIC language, reentry into the BASIC is achieved by pressing **CONTROL C**. This prevents your BASIC program from being damaged.

## 1.4 Correction of Errors While Inputing a Program

While inputing a program, the mistake in a line of statement can be corrected by pressing the BACKSPACE key <- before the carriage return key **↵** is pressed. Every press on the <- key deletes a character to the left of the cursor and back-space the cursor one position.

Example:

```
@PRIXT^
```

If we press the BACKSPACE key for two times, then we have

```
@PRI^
```

Now, we can key in the correct characters and get

```
@PRINT
```

## 1.5 BASIC Commands and Statements

Before discussing the BASIC commands and statements in the MPF-IP, we would describe the execution modes in the MPF-IP.

## 1.5.1 Execution Modes

In the MPF-IP, two execution modes are available. One is called the immediate execution mode, while the other deferred execution mode.

### 1) The Immediate Mode

The immediate mode allows a user to execute a BASIC command immediately after the command is entered.

Try to type on the keyboard as follows:

```
@PRINT 44/2
```

and then press , the MPF-IP will display.

```
22^
```

Which is the quotient of 44 being divided by 2.

From the example, it is obvious that after we entered the PRINT statement and press , the MPF-IP will execute the statement immediately to calculate the expression after the PRINT command and display the result on the indicator panel. In this example, "/" is the divide operator.

Now, try to type in the following statement line.

```
@PRINT 447*2;9*600+2*19
```

and press . What will you see on the display?

```
894      5438^
```

Isn't it?

### 2) The Deferred Mode

In the deferred mode, each BASIC statement is preceded by a statement number. Each statement is not executed immediately after it is entered. In the deferred mode, the statements contained in a BASIC program is not executed until the RUN command is given.

Now you are suggested to follow the example below to get a hands-on experience of how the deferred mode is.

```
@10 PRINT 447*2
@20 PRINT 9*600+2*19
```

Don't forget to end each statement line with .

A number of statements in this Deferred Mode will constitute a "program" which will not be executed immediately. Instead, they are stored in the memory. In fact, BASIC Interpreter will not execute this program until the user enter the command in the immediate mode RUN. Then the execution of the program will go on and will stop only after the whole program is performed. We will go into more detail on this subject in 4.2 "END/STOP statements".

Now type RUN and press the  key and we will have

894^

Press the  key again and we will see

5438^

Now if we press the  key for another three times, we will have the following displays in sequence:

```
READY
@
```

For more details on the function of the  in the execution of a program, see Chapter 9, : "Input/output statements".

## 1.5.2 Commands

The main functions of MPF-IP BASIC commands are to control the Editing, Executing and Debugging in a program.

Generally speaking, all the MPF-IP BASIC commands are executed in the immediate mode. But they can be executed in the deferred mode on some occasions. Similarly the deferred mode statements in MPF-IP BASIC can be converted to immediate mode to facilitate debugging.

One can enter any MPF-IP BASIC commands when the prompt character is displayed on the Indicator Panel. If there is any error while entering BASIC commands, the Indicator Panel will show an error message.

### 1.5.3 Statements

Every BASIC program is composed of a set of deferred mode statements aligned in order. More precisely, they are aligned according to their statement numbers, from the smallest to the largest. Statement numbers used in a program are restricted to integers between 1 and 9999. Leading zeros at the beginning of a statement number will be ignored in BASIC. For example, 012, 0012 and 12 will all mean the same thing 12 in BASIC. The users need not bother to enter statements in the order of statement numbers for BASIC will automatically align them in order. During the execution of a program BASIC will proceed in the order of the statement numbers unless one of the following situations develops

- (1) Commands related to the flow of the control are executed.
- (2) Execution is interrupted as a result of errors.
- (3) CONTROL and C are pressed simultaneously.
- (4) Execution proceeds to STOP or END statements.
- (5) After the statement with the largest statement number is executed.

During the entry of a program, if a statement number smaller than 1 or larger than 9999 is encountered, the Indicator Panel will display

SN ERROR IN LINE

If we press the  key three times now, we will see the following displays shown in sequence

READY  
@

And the following statements will not be accepted.

For Example:

```
@0 I=9
SN ERROR IN LINE
```

```
READY
@10000 K=8
SN ERROR IN LINE
```

```
READY
@01000 J=9
SN ERROR IN LINE
```

```
READY
@
```

In the last example above, 01000 does not exceed the allowed range of statement number, but it is regarded as an error because the length of this number exceeds four.

Free format is adopted in the MPF-IP BASIC statements. For example, the following statements all mean the same thing.

```
@20 LET B7=25
@20LET B7=25
@20LETB7=25
@20LETB 7=25
@20LETB 7= 25
```

however,

```
@20LET B7=2 5
```

is a statement with syntax error.

#### 1.5.4 Correct or Delete a Statement

##### \*Correct a statement

Correction is required on the following two occasions:

1. There is syntax error in the statement.

2. The whole program is required to be modified for a certain purpose.

Methods:

1. Before the  key is pressed, correction can easily be carried out with the <- key.
2. If the  key is already pressed:
  - (1) key in a new statement.

If the statement number of the new statement already exists somewhere in the original program, the old statement will be completely replaced by the new statement.

```
READY
@20 B7=25
@20 B7=12
```

- (2) Use the EDIT command

If only a minor part of the statement is to be corrected and the statement is a big one, (1) will take much time and efforts. In this case we can do it as follows:

```
@30 FOR I=1 TO 15 STEP 3
@EDIT 30,3,5
```

For more detail about the EDIT command, see 3.2.3.

**\*Delete a statement**

If you want to delete a statement, all you have to do is to key in its statement number and press the  key.

For example:

```
@20
```



## 1.6 Listing of a Program

After MPF-IP entered the BASIC, we can use the LIST command to list the statements stored in the computer.

The LIST command is used to list a program after correction or deletion of statements to see if the modification is done successfully as required.

For example:

```
@10 INPUT A,B
@20 C=A+G
@20 C=A+B:REM A CHANGE
@30 PRINT
@40 PRINT A;" +";B;" =";C
@LIST
```

```
10 INPUT A,B
20 C=A+B:REM A CHANGE
30 PRINT
40 PRINT A;" +";B;" =";C
```

The LIST command will list the statements starting from the statement with the smallest statement number to that with the largest statement number even if the statements were not entered in this order.

For example:

```
@40 PRINT A;" +";B;" =";C
@20 C=A+B
@30 PRINT
@10 INPUT A,B
@LIST
```

```
10 INPUT A,B
20 C=A+B
30 PRINT
40 PRINT A;" +";B;" =";C
```

```
READY
@
```

## 1.7 Execution of a Program

After a program is entered, we can type in RUN to request the execution of the program. THE LIST command can be used to see if the program is correct before execution.

For example:

```
@NEW:REM CLEAR PROGRAM
@100 PRINT "5*10=";5*10
@RUN
5*10=50

READY
@

@NEW
@10 INPUT "YOUR NAME: ",N$
@20 ? "HELLO! ";N$
@RUN
YOUR NAME:?CHENG
HELLO! CHENG

READY
@
```

In the above program, the REM statement is used as a remark, these statements are ignored in the execution of a program.

The INPUT statement is a statement which asks for entry (i.e., the user is expected to type in something through the keyboard) "?" is one of the output statement functioning exactly in the same way as PRINT. We will look over these statements in more detail later.

## 1.8 Deletion of a Whole Program

If an entered program is not to be used any more, the NEW command will delete them once and for all. In fact, the entry of the NEW command will nullify all the programs in the BASIC system. We have shown the usage of the NEW command in 1.7 though we did not go into details about it.

Now, try the following program:

```
@NEW
@100 PRINT "5*10=";5*10
@RUN
5*10=50

READY
@10 INPUT "YOUR NAME:",N$
@20 ? "HELLO! ";N$
@RUN
YOUR NAME:?YIH-HWA
HELLO! YIH-HWA
5*10=50
READY
@
```

In the above example, you will see the results as shown if you did not use the NEW command after the execution of statement 100, i.e., following the execution of statement 10 and 20, BASIC will proceed to the statement 100. Now, let us use the LIST command to examine what is stored in the computer.

```
@LIST
10 INPUT "YOUR NAME:",N$
20 ? "HELLO! ";N$
100 PRINT "5*10=";5*10

READY
@
```

## 1.9 Remark in a Program

In a program, we can use the REM statement to explain or interpret the function of the statements. When a program is displayed with a LIST command, the REM statement is also shown, but their presence will make no difference at all to the execution of a program.

For example:

```
@10 INPUT A,B
@20 C=A*B
@30 PRINT
@40 ? A;" *";B;" = ";C
@5REM...THIS PROGRAM
@7REM MULTIPLY 2 NUMBERS
@15REM 2 VALUES MUST BE
@17REMINPUT
@35REM...C CONTAINS THE
@37REM.....PRODUCT
@
```

As far as the execution of a program is concerned, the results will all be the same before and after the REM statements are added. Incidentally we would like to call your attention to "!". In fact, "!" is another kind of the REM statements functioning exactly in the same way as REM. This is just the same case with "?" and PRINT we mentioned earlier.

## 1.10 Usage of:

Earlier in 1.7 we have explained how to use ":" in the immediate mode together with the REM statement. Here let us look at its real power in practice more closely.

Except for its presence in a string, ":" denotes the end of a command or statement, followed by the beginning of another statement or command. This makes it possible for us to put several statements after a single statement number.

For example:

```
@10 FOR I = 1 TO 4 : ?I : NEXT I
1
2
3
4

READY
@10 FOR I = 1 TO 4
@20 ?I
@30 NEXT I
```

The above two programs will have the same result.

---

## Chapter 2 Expression

---

An expression is composed of constants, variables, functions and operators. An expression is expected to produce a certain value after evaluation. It can be a numeric value or a string. For that matter, an expression is called a numeric expression or a string expression according to the outcome to be produced by the expression.

Constants, variables and functions all denote a certain value. In an expression, the operators tell the computer what to do with these values. We will go into more detail on the evaluation of numeric values and strings in chapter 6 and 7. The following is an example of expression.

$(A+B) * C + D$

Here A,B,C, and D are variables, each one must be assigned a value in due course. In an expression, those included in parentheses are given top priority for execution, followed by multiplication and division, and lastly addition and subtraction are executed.

If A=6, B=4, C=2, D=1, the value of this expression is 21.

See the next expression:

$(X*(Y-2))+Z$

Here, X,Y,Z are variables supposedly assigned certain numeric values, \*,+,-, are symbols of multiplication, addition and subtraction respectively. 2 is a constant. Those nested in the innermost parentheses are evaluated first, followed by those in the outer parentheses and lastly the operation of addition. If X,Y,Z are 7,4,3 respectively, the value of this numeric expression is 17.

## 2.1 Constants

A constant can either be a numeric value or a string.

### 2.1.1 Numeric Constants

A numeric constant can be a positive number, or a negative number, or zero. In MPF-IP BASIC, a numeric constant can be a real number or a number expressed in scientific notation. Both are expressed in floating point system in MPF-IP BASIC. As a result, during the execution of a program, the number displayed on the indicator panel after PRINT or "?" statements as well as those stored in the computer memory are subject to the following limitations.

Inside the computer, real numbers and numbers expressed in scientific notation are the same in essence. In fact both are expressed in floating point system, and can be considered the same thing. But in the design of MPF-IP BASIC, the precision is restricted to 6 digits. Therefore, any number greater than the 6th power of 10 or less than the -6th power of 10 can not be expressed with 6 digit precision. In this case, the scientific digit precision. In this case, the scientific notation is used as the key to solve to this problem.

A real number can be any one between 999999 and -999999, with or without a decimal point, but the number of digits shall not exceed 6. If the exponent of a number is larger than 6 or smaller than -6, BASIC will automatically express it in scientific notation.

The scientific notation makes it easy to deal with numbers with very small or very large exponent. The format of the scientific notation is shown below:

Number = Mantissa \* 10<sup>Exponent</sup>

Examples:

| Conventional Scientific Notation | MPF-IP BASIC Format |
|----------------------------------|---------------------|
| 3.264 * 10 <sup>4</sup>          | 3.264E4             |
| 9.99 * 10 <sup>17</sup>          | 9.99E17             |
| 5.691 * 10 <sup>-5</sup>         | 5.691E-5            |
| -2.47 * 10 <sup>-18</sup>        | -2.47E-18           |

If the absolute value of a number is greater than 4.9999E+18 or less than -9.9999E-20, then the number can not be expressed in scientific notation.

For example:

```
@PRINT 4.5624E5
456240
@PRINT 234.5678E9
2.34567E+11
@PRINT 3E7
3.00000E+07
```

Underflow:

When a small number falls between -9.99998E-20 and 9.99998E-20, there will be underflow error with the following error message displayed:

```
UN ERROR IN LINE nnnn
```

and the program will go on.

Overflow:

If a large number goes above 4.99999E+18 or falls below -4.9999E+18, there will be Overflow error with the following message displayed:

```
OV ERROR IN LINE nnnn
```

and the program will go on.

For example:

```
@PRINT 1E-19
9.99998E-20
@PRINT 1E-20
UN ERROR IN LINE
7.99998E-20
@PRINT 5E18
4.99999E+18
@PRINT 1E19
OV ERROR IN LINE
4.99999E+19
@PRINT 3E19
OV ERROR IN LINE
3.74999E+18
@NEW
@10 I=2
@20 T=I*I
@30 PRINT I
@40 GOTO 20
@RUN
4
16
256
65536
4.29496E+09
OV ERROR IN LINE 20
2.30584E+18
OV ERROR IN LINE 20
2.30584E+18 CONTROL C 
READY
@
```

### 2.1.2 Literal Strings

Literal strings are strings of ASCII characters enclosed by two quotation marks. In a literal string, a space does mean something. In fact, it is considered as a character. This is especially important for those who are accustomed to free format.

We can put any of the characters that can be entered through the keyboard into a literal string. Due to the limited size of the key-in buffer, the number of characters in a literal string is restricted.



In fact, the size of the key-in buffer is 60 characters, but the number of characters in a literal string can be as large as 255 in actual operations.

Example:

```
"CBABC"  
"WHAT A MAN!!"  
"CHENG YIH-HWA TEL 894-5438"  
"HAHA"
```

Due to the requirement of the system design, there are occasions we have to appeal to special ways to have a character displayed. We will go into more detail in this respect in chapter 8 on CHR\$. For now, let us try the following example:

```
@PRINT "ABC"+CHR$(13)+"HAHA"  
ABC  
HAHA
```

## 2.2 Variables

A variable is a name with an assigned value. During the execution of a program the assigned value is subject to changes on request of the programmer.

The variable accounts for a major part in a program. Each time the name of a variable is referenced, it is the contents (a constant) of that variable that is accessed.

The name of a numeric variable can be a letter of the alphabet (A-Z) or an alphabet followed by numeric letter (0-9). The value of a numeric variable is a floating number. Here are some examples:

```
A    A0  
E    E5  
Z    Z9
```

The name of a string variable can be obtained in the same way as a numeric variable except that it must be added with a "\$" at the end. Here are some examples:

|     |      |
|-----|------|
| A\$ | A0\$ |
| E\$ | E5\$ |
| Z\$ | Z9\$ |

Usually, the value of a string variable is a string of characters. In an extreme case, it can be a null string, i.e., the length of the string is zero.

If the name of a variable is to represent an element of an array, the variable is expected to be added with a subscript. A variable with a subscript is the same as a common constant in essence. The number of subscripts to variables can be one or two. If there are 2 subscripts, they must be separated by a comma(.). In both cases, subscripts are enclosed by parentheses. Usually, the subscripts are numeric constants or numeric variables, but there are chances that they are numeric expressions. As the value of a numeric expression is a floating number, they will be rounded up during the evaluation when they are used as subscripts. Here are some examples:

|        |            |
|--------|------------|
| A(1)   | A0(A2,A3)  |
| E(1,2) | E5(C4,X/4) |
| E(N+1) | Z9(10,10)  |

A string variable cannot be an element of an array.

## 2.3 Functions

A function is an operation in which a single value is obtained through a series of evaluations of one or more parameters. A numeric function has the same notation with a numeric array. But the number of parameters in a numeric function is not restricted to one or two. The parameters are also separated by commas. The number of parameters and type are different from one function to another.

For the simple reason that the outcome of a function after evaluation is a certain value, a function can be used anywhere in expressions just like variables and constants. In practice, a function is present in expressions in the format of a function name followed by an actual parameter enclosed in parentheses.

Let us look at some examples.

INT(X):

Here X is a numeric expression. The outcome after evaluation from this function is the greatest integer smaller than or equal to X. For instance:

INT(8.35)=8

SGN(X):

Again, X is a numeric expression. We will have the outcome according to the following definition:

1, if  $X > 0$   
0, if  $X = 0$   
-1, if  $X < 0$

For instance,

SGN(4 \* -3)=-1

MPF-IP BASIC provides the user with quite a number of built-in functions, such as the sine function, the square root function, the function to get the absolute values, etc. In Chapter 6, we will examine these useful numeric functions in great detail. They are listed in Appendix D. In addition, if one has to use the same series of operations to get a value frequently, it is possible for him to resort to user-defined functions. For more detail please see chapter 11.

So far we have focused our description on numeric functions. When the outcome of a function after operation turns out to be a string of letters of the alphabet, it is called a string function. See chapter 8 for more detail. Again, some built-in string functions are listed in Appendix D.

## 2.4 Operators

The operators will work arithmetic, logic or string operations on one or two values to yield an outcome. Usually, an operator comes between two operands, and is called a Binary Operator. There are, however, Unary operators as well. For instance, the "-" is binary in A-B, but is unary in -A. The combinations of operands and operators bring about expressions. The operands in an expression can be constants, variables, functions, or another expression. The operators can be classified according to their characteristics into two categories: (1) arithmetic and (2) relational operators.

### (1) Arithmetic Operators

#### (A) Unary

|   |                   |    |
|---|-------------------|----|
| + | the positive sign | +A |
| - | the negative sign | -A |

#### (B) Binary

|   |                |     |
|---|----------------|-----|
| + | Addition       | A+B |
| - | Subtraction    | A-B |
| * | Multiplication | A*B |
| / | Division       | A/B |
| ↑ | Exponentiation | A↑B |

### (2) Relational Operators

|    |                          |      |
|----|--------------------------|------|
| =  | equal to                 | A=B  |
| <  | smaller than             | A<B  |
| >  | larger than              | A>B  |
| <= | smaller than or equal to | A<=B |
| >= | larger than or equal to  | A>=B |
| <> | unequal to               | A<>B |

In an operation with relational operators, a logical value will come out as the result. In practice, all relational operators are binary and relational expressions can be used only in IF statement.

### Operation of strings

In MPF-IP BASIC, there is an operator for the strings used to combine 2 strings. The corresponding operation is called concatenation.

| notation | meaning       | example   |
|----------|---------------|-----------|
| +        | concatenation | A\$ + B\$ |

Here, A\$ and B\$ are combined to form a new string. In the new string, the part of the original A\$ is immediately followed by B\$. For instance, if A\$="ABC":B\$="DEFG" then A\$+B\$="ABCDEFGG". (A more detailed discussion concerning the operation of strings will be found in Chapter 8.

## 2.5 Evaluating Expressions

In essence, to evaluate an expression is to obtain the value of each part of an expression and get the final value after the operations by the operators. Following is the outline for evaluating expressions:

- (1) Substitute the value of a variable for the variable
- (2) Implement the operation as defined by the functions to get the result value of the function.
- (3) Implement the operations indicated by the operators

In evaluating expressions, operators are evaluated according to their priorities.

|                 |                                     |
|-----------------|-------------------------------------|
| Higher priority | Unary +,-<br>↑<br>*,/<br>binary +,- |
| Lower priority  | relational operators =,<,>,<=,>=,<> |

The evaluation of expressions starts with operations with higher priorities and proceeds to those with lower priorities as shown in the above list. Should there be two operators of the same priority, the evaluation will be in the order from left to right. But the operations enclosed in parentheses will be carried out in the first place, overriding all the orders described in the above. In other words, operations enclosed in parentheses are always evaluated earlier than those outside of the parentheses. In case there are multifold parentheses, the innermost nested parentheses is given the top priority.

Let's look at some examples:

$7+5*6$  and  $7+(5*6)=37$  are the same thing.

$6/2*4/8$  is equal to  $((6/2*4)/8)=1.5$

Provided  $A=4$ ,  $B=5$ ,  $C=6.1$ ,  $D=0$ ,  $E=4.3$

then  $A+B*C=4+(B*C)=34.5$

$A*B-C=(A*B)-C=13.9$

$A-B-C=(A-B)-C=7.1$

$(A+B)*C=54.9$

In a relational expression, relational operators are used to obtain logic value (True or False) of an expression. If  $A, B, C, D, E$  are assigned the same values as above, then the logic value of the expression

$(A*B) < (D+E)*C$

is True. Here  $(A*B)=20$ ,  $(D+E)*C=26.23$

---

## Chapter 3 Commands

---

So far we have described the usage of the commands LIST, RUN and NEW. Parameters can be put after the LIST command, which was not mentioned in earlier chapters. In this chapter, we will present to you all the commands used in MPF-IP BASIC and you can find a list of them in Appendix C.

The commands used in MPF-IP BASIC are classified according to their characteristics into 4 categories, i.e., (1) Execution Commands, and (2) Editing Commands and (3) Storing Commands and (4) Auxiliary Commands.

### Execution Commands:

```
RUN
XEQ
GOTO
CONTINUE
QUIT
```

### Editing Commands:

```
LIST
NEW
NEW*
EDIT
```

### Storing Commands:

```
LOAD
SAVE
```

## Auxiliary Commands:

FREE  
HEX

### 3.1 Execution Commands

The execution of a BASIC program could be suspended through program design (e.g., by setting break points in the program) or by pressing CTRL-C or RESET. When the execution of a program is suspended, we can examine the execution state using some commands or statements. In practice, we can debug a program at the break point, e.g., retrieve the value of a certain variable, change it and then continue the execution from the break point. This is also the case with the whole program. You can list the whole or a part of a program, change it and then continue the execution.

#### 3.1.1 RUN/XEQ/GOTO

RUN - Execute a Program

Format:

RUN

Execution Mode:

Immediate & deferred

Description:

The RUN Command is used to execute an MPF-IP BASIC program.

Usually execution of a program starts from the smallest statement numbers.

Remark:

RUN is used to execute a program, and prior to the execution, the BASIC will "clear" the contents of all the variables. i.e., the numeric variables are initialized with the value 0, and the string variables are reset to be null strings.



XEQ - The execution of a program.

Format:

XEQ

Execution mode:

Immediate & deferred

Description:

XEQ is also used to execute an MPF-IP BASIC program somewhat the way the RUN command is used. But they are different when it comes to the initial values of the variables in a program.

Remark:

XEQ is different from RUN in that XEQ will execute a program without incurring a "clearing" of the contents of variables prior to the execution. In other words, the values of the variables set earlier will not be affected by the execution of the XEQ command.

GOTO - Changing the starting point of execution of a program.

Format:

GOTO statement number

Description:

GOTO command is also used to execute a program. But the starting point of execution is the statement number specified in the command, not the smallest statement number in the program.

Note:

If the statement number specified in the command is not to be found anywhere in the original program, you will see the following error message displayed in the indicator panel:

UL ERROR IN LINE

and the execution of the program is suspended.

Remark:

Compared with RUN, GOTO is similar to the XEQ command to a greater extent for the execution of GOTO does not affect the value of the variables in a program. In other words, the GOTO command does not cause initialization. For this reason, GOTO is used mostly for debugging.

To sum up, when we make comparisons among the three commands RUN, XEQ and GOTO, we will find that RUN is used in regular execution, XEQ is used when a specific purpose in a program is to be achieved in mind and GOTO is used mostly for debugging. As you have noticed, the starting point of execution in a program is not necessarily from that with the smallest statement number. Let us try the following examples:

```
@10 PRINT"A=";A;"B$=";B$
```

```
@20 IFA<>0 THEN40
```

```
@30 B$="HA!HA!"
```

```
@40 C=20:D=2:A=C^D
```

```
@50 STOP
```

```
@60 B$="I AM FINE"
```

```
@70 PRINT B$
```

```
@80 A=0
```

```
@RUN
```

```
A=0 B$=
```

```
STOP AT LINE 50
```

```
@XEQ
```

```
A=399.998 B$=HA!HA!
```

```
I AM FINE
```

```
READY
```

```
@XEQ
```

```
A=0 B$=I AM FINE
```

```
STOP AT LINE 50
```

```
@XEQ
```

```
A=399.998 B$=HA!HA!
```

```
I AM FINE
```

```
READY
```

```
@GOTO70
```

```
I AM FINE
```

```

READY
@XEQ
A=0 B$=I AM FIN

STOP AT LINE 50
@RUN
A=0 B$=

STOP AT LINE 50
@GOTO 65
UL ERROR IN LINE

READY
@

```

In the above example, RUN is used twice and XEQ four times. As to the usage of GOTO, one is correct, but in the other one, the specified statement number 65 is not found in the program, so an error message is displayed. You may have noticed the implementations of the RUN commands have resulted in the same outcome. This is because at the beginning of the execution of the program all numeric variables are cleared to 0 and strings variables null. In the case of XEQ, however, different outcomes are obtained because the value of A and B\$ are not cleared to 0 or null.

You will notice during the execution that after the first RUN command, we get  $A=C^D=399.998$  ( $A=20^2=20 * 20=400$ ). For more detail please see chapter 6), therefore by the first XEQ you have a different outcome with that of the RUN command. Later, after the implementation of statement 80  $A=0$ , the outcome of the second XEQ is obtained.

### 3.1.2 Continue

CON - Resume execution of an interrupted program.

Format:

```
CON[TINUE]
```

Execution Mode:

Immediate

### Description:

CON is used to resume execution of a program interrupted as a result of a simultaneous press on CONTROL and C or a STOP statement.

Let us look at the following examples:

#### Example 1.

```
@10 FOR I = 1 TO 6
@20 PRINT I, : IF I=3 THEN?
@30 NEXT I
@40 PRINT:PRINT"*DONE*"
@50 STOP
@60 PRINT "***RESUME***"
@RUN
  1          2          3
  4          5          6
*DONE*

STOP AT LINE 50
@CON
**RESUME**

READY
@
```

#### Example 2.

```
@10 FOR I=1 TO 100: ?I next I
@RUN
  1
  2
  4
  ↓ C
STOP AT LINE 10
@CON
  6
  7
  8
  ↓ C
STOP AT LINE 10
```

### 3.1.3 QUIT

QUIT - Return control to monitor program

Format:

QUIT

Description:

QUIT is used to exit BASIC language and return control to the monitor program. The QUIT command can be executed either in immediate or in deferred mode.

```
READY
@XEQ
A=0 B$=I AM FIN

STOP AT LINE 50
@RUN
A=0 B$=

STOP AT LINE 50
@GOTO 65
UL ERROR IN LINE

READY
@
```

In the above example, RUN is used twice and XEQ four times. As to the usage of GOTO, one is correct, but in the other one, the specified statement number 65 is not found in the program, so an error message is displayed. You may have noticed the implementations of the RUN commands have resulted in the same outcome. This is because at the beginning of the execution of the program all numeric variables are cleared to 0 and strings variables null. In the case of XEQ, however, different outcomes are obtained because the value of A and B\$ are not cleared to 0 or null.

You will notice during the execution that after the first RUN command, we get  $A=C^D=399.998$  ( $A=20^2=20 * 20 = 400$ ). For more detail please see chapter 6), therefore by the first XEQ you have a different outcome with that of the RUN command. Later, after the implementation of statement 80  $A=0$ , the outcome of the second XEQ is obtained.

## 3.2 Editing Commands

Editing commands are used to modify or process the program currently stored in the BASIC system. There are 3 editing commands: LIST, NEW[\*] and EDIT.

### 3.2.1 LIST

LIST - Listing a program.

Format:

```
LIST [[n][,m]]  
(n,m are statement numbers)
```

Description:

There are a variety of derived forms among the LIST commands as follows.

```
LIST      List the whole program  
LIST n    List the statement with statement number n  
LIST n,m  List the statements with statement numbers  
           from n to m.  
LIST ,m   List the program from the beginning to the  
           statement to that with the statement number m.
```

Execution Mode:

Immediate execution mode.

Note:

In LIST n,m, no statements will be listed if  $n > m$ .

Remark:

LIST ,m, an equivalent of LIST 0,m, can be considered as a special case of the LIST n,m form. This is an implication of the fact that all statements with statement number larger than n and smaller than m will be listed, even if n is not found in the current program. We do not give you the LIST m, form because no statements will be listed as it is equivalent to LIST m,0.

```
@10 A=40
@20 B=25
@30 C=A*B
@40 PRINT C
@LIST
10A=40
20B=25
30C=A*B
40PRINT C
```

```
READY
@LIST,30
10A=40
20B=25
30C=A*B
READY
@LIST 30,40
30C=A*B
40 PRINT C
```

```
READY
@LIST11,25
20B=25
```

```
READY
@LIST20,
```

```
READY
@LIST30
30C=A*B
```

```
READY
@
```

### 3.2.2 NEW/NEW\*

NEW/NEW\* - Clear a program

Format:

NEW[\*]

Description:

There are two forms of the NEW command as follows:

- (1) NEW -Clears the program as well as all the variables
- (2) NEW\*-Clears the program only.

Execution Mode:

Immediate & deferred

Remark:

When NEW\* is executed, only the program is cleared, all the variables are left with their original values unaffected.

Try the following example:

```
@10 I=5
@RUN
```

```
READY
@NEW*
@10 PRINT
@XEQ
5
```

```
READY
@NEW
@10 PRINT
@XEQ
0
```

### 3.2.3 EDIT

EDIT - Modify a statement.



Format:

EDIT n/string-1/string-2

Description:

Replace string 1 in statement n with string 2.

Execution Mode:

Immediate execution mode.

Note:

When statement number n is not found in the program, the execution of EDIT will cause the display of the following error message:

UL ERROR IN LINE

Similarly, if string 1 is not found in statement n you will observe:

DA ERROR IN LINE

Remark:

We have used two slants "/" as delimiters to separate the two strings. In fact, any two identical characters can be used in pair as delimiters. The character, however, must not be a space.

For instance:

@10 A=7\*5/4+3

@EDIT 10/7/6

@LIST

10A=6\*5/4+3

READY

@EDIT 10?5/?3-

@LIST

10A=6\*3-4+3

READY

@EDIT 15V6V3

UL ERROR IN LINE

```
READY
@EDIT 10 3 4
SN ERROR IN LINE
```

```
READY
@EDIT 10VxVI
DA ERROR IN LINE
```

```
READY
@
```

### 3.3 Permanent Storage Commands

After a program is completed, it can be stored onto a magnetic tape for use in the future. After the storage operation, the program is still available in the system. It will not be cleared off until the a NEW command is implemented.

To store a program onto a magnetic tape, one must use the SAVE command followed by a 4-character program name.

#### 3.3.1 SAVE

SAVE - Store a program onto a magnetic tape.

Format:

```
SAVE aaaa
(a represents any character except space)
```

Description:

To permanently store a MPF-IP BASIC program, we can use the SAVE command in conjunction with a recording device to store the program onto the magnetic tape. In the format, aaaa is the program name.

Execution mode:

Immediate Execution mode

#### Note:

Any character following SAVE must not be a space, otherwise, the program will be read only with extreme difficulty. The program name must be composed of exactly four characters.

Suppose we have the following program:

```
@NEW
@10 I=2
@20 I=I*I
@30 PRINT I
@40 GOTO 20
```

First, set the recording device ready for use, i.e., connect one end of the recording cable to the MIC jack of the recorder and connect the other end to the MIC jack of MPF-IP. Then type in:

```
@SAVE POWR
```

Be sure to leave room for a space after SAVE. Now press the REC and PLAY on the recorder followed by a press on the  key. As this is done you will see:

```
@
```

### 3.3.2 LOAD

LOAD - Read a program from the magnetic tape.

Format:

```
LOAD aaaa
(a is any character except space)
```

Description:

The LOAD command is used to read a program which was previously written onto the magnetic tape by the SAVE command back to the BASIC system.

Execution mode:

Immediate execution mode

Note:

All the 4 characters following LOAD are expected to be non-space characters.

Let us see how the LOAD command works.

First, type in

```
@NEW
```

Now connect the EAR jack of the recorder and that of the MPF-IP with a connecting cable. Then type in

```
@LOAD POWR
```

And now press the  key and the PLAY button on the recorder in sequence. You will see the following display on the indicator panel.

```
....  
POWR  
----
```

```
@LIST  
10 I=2  
20 I=I*I  
30 PRINT I  
40 GOTO 20
```

```
READY  
@
```

### 3.4 Auxiliary Commands

In MPF-IP BASIC, two auxiliary commands are available to provide useful information as needed. One of the is used to tell the user how many memory locations are still usable, and the other can convert the hexadecimal into the decimal representation.

### 3.4.1 FREE

Format:

FREE

Description:

After the execution of the FREE command, two hexadecimal numbers in 2 lines will be displayed. The one in the first line shows the start address of free memory locations and the other shows how many memory locations are still free. It is always a good practice to guard against using the memory locations in an uneffective manner, especially when one is to enter a long program. For instance, you may well use "?" instead of PRINT and "!" instead of REM in a program.

Let us look at some examples of the FREE command.

```
@NEW
@FREE
F000
0B9E
@10 FOR I = 1 TO 4
@20 A(I)=I
@30 PRINT I,A(I)
@40 NEXT I
@FREE
F000
0B6E
@RUN
```

|   |   |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |

```
READY
@FREE
F000
097B
```

### 3.4.2 HEX

HEX - Convert the hexadecimal numbers into decimal.

Format:

HEX nnnn

Description:

The HEX command is used to convert a hexadecimal number into a decimal number and display it.

Execution mode:

Immediate execution mode.

Remark:

The auxiliary command HEX is quite often used in conjunction with the CALL statement. For more detail on the CALL statement, please see Chapter 12.

#### A Review on Execution Mode

In order to facilitate debugging, MPF-IP BASIC is designed so that all commands and statements can be executed both in immediate and in deferred execution mode. In this manual, we have classified the commands and statements by execution mode in a convention that most other BASIC adopted. It is true that all commands and statements can be executed in both modes. However, for some statements, it is only when they are executed in immediate mode in conjunction with other statements that we can make the best of them. For this reason, we have assigned each command and statement with their optimum execution mode.

Let us give a description of them as follows:

(1) Immediate Mode: In MPF-IP BASIC, when the prompt character "@" is displayed, any entry without a statement number will cause an immediate execution.

(2) Deferred Mode: In MPF-IP BASIC, any command or statement with a statement number will not be executed immediately. They will not be executed until the commands RUN/XEQ/GOTO is entered. Usually a BASIC program is composed of a number of deferred statements. When we run the program, BASIC will ignore the statement numbers and all statements are in a sense, executed in the immediate mode.

---

## Chapter 4

### General Statements

---

We will describe the various types of statements used in BASIC programs in the subsequent chapters. In this chapter we will show you the general statements. In 1.5.2 we have given you a rough description of statements, and you know a statement begins with a statement number and ends with a press on the  key. And in 1.10 we have told you that following a statement number we can put several statements separated by colons (":"). A program composed of statements will not run until the implementation of execution command. In order to facilitate debugging, most MPF-IP BASIC statements can be executed either in immediate or in deferred mode.

#### 4.1 LET

LET - Assignment statement.

Format:

[LET] Var = Expr.

Description:

With this statement, you can assign the value of Expr to the variable Var. Usually the types of the variables on the two sides of the equal sign must be the same.

Execution mode:

Deferred mode.



Remark:

If the types of variables on the two sides of the equal sign are different, the following error message will be displayed

SN ERROR IN LINE

Remark:

In this statement, the equal sign can be considered as an operator. It is, however, different from the equal sign used as a relational operator in that it assigns the value of the expression to the right of the equal sign to the variable on the left side of the equal sign rather than denotes equality.

In case the lefthand side variable happens to be an element of an array, the subscript of the variable will be evaluated in the first place and then the righthand side expression is evaluated

The following is an example:

```
@10 LETF=3.4
@20 R=555
@30 A$="DINGDONGDING"
@40 A1(2)=2
@50 A1(A1(2))=4
@60 ?F;R;A1(2)
@70 ?A$
@80 A$=R
@RUN
3.4 555 4
DINGDONGDING
SN ERROR IN LINE 80
```

READY

@

## 4.2 END/STOP

END and STOP statements are used to terminate the execution of a program. As far as the program execution is concerned, you can do with both of them or without either of them. As a matter of fact, in MPF-IP BASIC, there is implicitly an END statement at the end of the statement with the largest statement number.

END - Terminate the execution of a program.

Format:

END

Description:

The END statement is used to terminate the execution of a program. After it is executed, the following message will appear on the display:

READY

Execution mode:

Deferred mode

Remark:

The END statement does not necessarily appear at the last line of a program, it can be anywhere in the program.

STOP - Suspend the program execution

Foremat:

STOP

Description:

The STOP statement signifies a break point in a program, after it is executed, the following message will appear on the display:

STOP AT LINE

Execution mode:

deferred mode

Remark:

STOP is used to suspend the program execution in much the same way as END. It is however, different from the END statement in that after being executed, one can use the CON command to resume the program execution. Thus in a strict sense, STOP is merely a break point in the program.

Now try the following example:

```
@10 C=1:C1=C
@20 FOR I = 1 TO 10
@30 C=C+C1:C1=C
@40 NEXT I
@50 PRINT "C=";C
@60 STOP
@70 PRINT "CONTINUE"
@80 END
RUN
C=1024

STOP AT LINE 60
@CON
CONTINUE
READY
@
```

### 4.3 REM and "!"

The REM statement can be used anywhere in a program to insert remarks with a view to making the program easier to read. In a big program, the readability is of great importance. A program difficult to read will result in extreme impediment in program documentation and maintenance. Even the author of a program will understand his own program only with much pains if it was programmed long ago and written without sufficient remarks.

REM - Remarks in a program.

Format:

```
REM|![(CHARACTER)]
```

Description:

Any character in a REM statement will not affect the program execution in BASIC in any way. In other words, REM is a non-execution statement. The character "!" have exactly the same function as that of "REM". As any character in a REM statement is ignored by the BASIC, even ":" (As mentioned in 1.10, it is used to separate statements.) will not function as it does.

See the following example:

```
@10 REM:HERE SHOWS
@20 REM AN EXAMPLE
@30 REM OF REM STATEMENTS
@40 REM-ANY CHARACTERS
@50 REM-MAY FOLLOW REM
@60 REM-REM STATEMENTS
@70 REM-ARE NOT EXECUTED
@80 PRINT "REMARK"
@90 !HERE IS ANOTHER REM
@95 PRINT "SEE YOU AGAIN"
@RUN
REMARK
SEE YOU AGAIN

READY
@
```

## 4.4 RANDOMIZE

The RANDOMIZE statement is used to change the initial value of the random number function RND. (See chapter 6)

RANDOMIZE - Change the initial value of random numbers.

Format:

```
RAN(DOMIZE)
```

#### Description:

The RAN statement is used to renew the initialization of a set of random numbers.

#### Execution mode:

Deferred mode

#### Remark:

For more detail please see the section on the RND function.

## 4.5 SON (Speed On) and SOFF (Speed Off)

In the Basic mode, MPF-IP supplies users with two extra optional instructions to control the speed of execution of a program, with these two instructions, you can choose to execute your program at a faster speed or normal (built-in) speed. The rate of execution of "SON" is about four or five times of that of "SOFF". Default is "SOFF".

The reason why you may feel your program executes slower when using the instruction "SOFF" is that MPF-IP has to spend time scanning the keyboard and the display, even though no data is to be output to the display.

#### Description:

If one intends to speed up the execution of a certain block of instructions in a program (such as a block with arithmetic calculation), put the instruction "SON" with the statement number in front of that block of instructions. On the contrary, if one intends to go back to the normal (built-in) speed to execute a certain block of instructions in a program (such as a block sending information out to the printer or display of MPF-IP), put the instruction "SOFF" with the statement number in front of that block of instructions.

However, the speed of execution will be affected even with "SON", if you intend to print something out through the printer which is in "PRT-ON" status and connects to MPF-IP.

Format:

SON/SOFF

Execution Mode:

Deferred Mode

Please type in the following programs, and you will see what is the difference between these programs through the display of MPF-IP and time-counting.

1) A Do Loop preceded by a " SON " command:

```
10 SON
20 FOR I=1 TO 5000
30 NEXT I
@ RUN
```

Note: It takes 26 seconds to execute the above program.

2) A Do Loop preceded by a " SOFF " command:

```
10 SOFF
20 FOR I=1 TO 5000
30 NEXT I
@ RUN
```

Note: It takes 1 minute and 46 seconds to execute the above program.

3) Two Do Loops with " PRINT " statement Preceded by a " SON " and " SOFF " command respectively:

```
10 NEW
20 SON
30 FOR I=1 TO 33:PRINT I;:NEXT I
40 SOFF
50 FOR I=1 TO 33:PRINT I,:NEXT I
@ RUN
```

Remark:

If the "PRINT ," or "PRINT ;" statement is preceded by a SON command, data is only printed on the printer board if it exists and not displayed on the 20-digit FIP (Fluorescent Indicator Panel). If "," or ";" is not in the PRINT statement, data is printed on both the printer and the display (FIP).

---

## Chapter 5

### Control Statement

---

In this chapter, we shall investigate the control statements used in MPF-IP BASIC. The control statements are used in the following four applications:

- .LOOP
- .Conditional Control Transfer
- .Unconditional Control Transfer
- .Computed Control Transfer

#### 5.1 Loop

The loop is an indispensable tool in programming. The size of memory of a computer is restricted by physical and cost considerations. The aim of programming is to make the most out of this limited memory. For that reason, the recursive utilization of memory is, in a sense, the best way to extend the memory.

All that is stored in the computer memory can be classified into the following two categories:

- (1) Data and (2) Procedure.

The utilization of variable was developed out of the consideration to repeatedly utilize the data memory. This is again the case with the utilization of loop and procedure memory.



Generally speaking, the program execution follows a normal order, i.e., executed from the beginning to the last as indicated by the statement numbers one after another. In actual practice, however, most computer programs are not executed this way. Many problems require changes of execution sequence whenever necessary. On some occasions, some statements are skipped over while on other occasions certain statements are requested to be repeated. These in all bring the loop into existence. Usually a loop is composed of four components. However, not all loops consist of the four parts. The four components are described below:

#### (1) Setup

The setup of a loop requires at least the initialization of a control variable.

#### (2) Body of the loop

By "Body of the loop" we mean all the statements in the loop in general. Naturally, it may as well include the nested loop or loops.

#### (3) Modification of the control variables

The execution times of the whole loop is decided by the value of the control variables. As a consequence, if one cannot modify the control variables, the loop will turn into an Infinite loop.

#### (4) Test/Exit

Test/Exit is provided to determine if a loop is to carry on repeated execution. The contents of Test is the execution factor controlled by the control variable.

In the section on Loop, we will describe FOR/NEXT. In the section on Conditional Control Transfer, we will investigate the recursive loops formed by IF/GOTO. In the section on Unconditional Control Transfer, we will see a special form of loops - the Infinite Loop. In the last pages we will show you the applications of Computed Control Transfer.

### 5.1.1 FOR

FOR - the FOR loop

Format:

FOR avar = aexpr-1 TO aexpr-2 (STEP aexpr-3)

Description:

The FOR statement is used to form a loop. In analysis, we find avar aexpr-1 forms the Setup component, (STEP aexpr-3) is used to modify control variable, the control variable is avar, and TO expr-2 is the TEST part of TEST/EXIT.

Execution mode:

Deferred Mode

Remark:

To ensure the completeness of a loop, a FOR statement has to be accompanied by a NEXT statement. In this case, the NEXT statement functions as the EXIT part of TEST/EXIT. All that is inclosed by FOR and NEXT forms the Body of the loop. As is seen in the format, the part STEP aexpr-3 can be omitted, in this case, there is implied STEP 1.

### 5.1.2 NEXT

NEXT - the next execution reptition of the FOR loop.

Format:

NEXT avar-2

Description:

The FOR statement is the start point of a FOR loop, while the NEXT statement is the end point of it.

Execution mode:

Deferred mode

#### Remark:

The FOR statement and the NEXT statement are combined to form a FOR loop. In this case, avar-1 in FOR must be identical with avar-2 in NEXT. In 1.10, we described the usage of ":", here we would like to tell you that it is illegal to use ":" followed by another statement after the NEXT statement.

### 5.1.3 FOR/NEXT LOOP

The loop made up of the combination of FOR/NEXT is used to repeatedly execute a group of statements. The statements group starts with the FOR statement and end at the NEXT statement. The number of repeated execution is decided by other parts in the FOR statement. As a review, please refer to 5.1.2. In this section we will continue the description in more detail.

During the execution of the FOR statement, avar serves as the control variable of the loop. The loop setup is accomplished when aexpr-1 is evaluated and assigned to the control variable as its initial value. The subsequent process is described below:

(1) First a comparison between control variables avar and expr-2 is made, when aexpr-3 is positive, if  $avar > aexpr-2$  the program execution will jump to the one immediately following the NEXT statement. When aexpr-3 is negative, if  $avar < aexpr-2$  the execution will go on from the one immediately following the FOR statement.

(2) The execution of the statements group specified by the FOR and the NEXT statement will then continue.

(3) The control variable is updated. If step aexpr-3 in the FOR statement is omitted, the control variable is incremented by 1, otherwise, it is incremented by the value specified by aexpr-3.

(4) The program execution again goes back to (1).

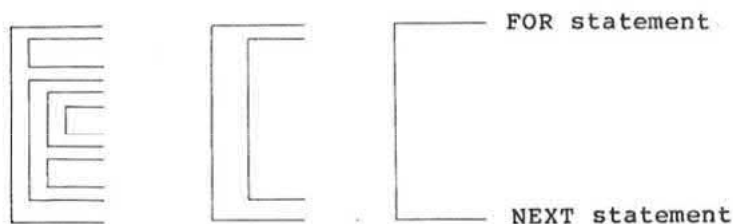
Each time a FOR loop begins execution, the MPF-IP BASIC will first examine if there is another FOR loop, If so, and if the control variables of the FORINEXT loops are identical, then the FOR loop originally in existence will be ignored and disabled, and the program execution will go on as stated before.

It is a good practice during programming to avoid making program execution directly jump to the body of the loop without an appropriate setup of the FOR loop beforehand. Otherwise some unpredictable results may occur.

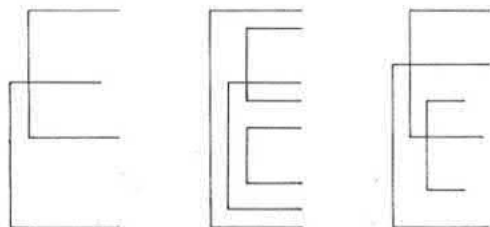
There can be FOR loops in nested structure. An Inner Loop is one that is completely contained in the body of the Outer Loop. Overlapping of two loops is not allowed.

In order to make you understand the correct usage of loops in nested structure and control transfer (for more detail on this topic see the following pages of this chapter) and FOR loop, some examples of legal and illegal structure are listed below for your reference.

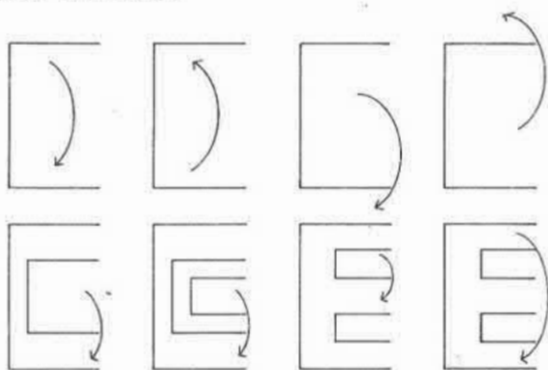
Legal Loops in Nested Structure:



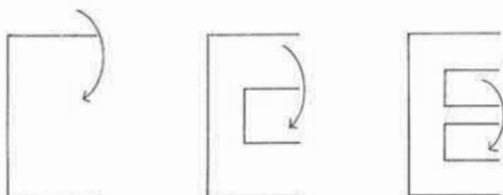
Illegal Loops in Nested Structure



## Legal Control Transfer



## Illegal Control Transfer



### 5.1.4 Some examples

```
@NEW
@10 FOR I = 1 TO 6
@40 PRINT 1/I
@80 NEXT I
@RUN
1
.5
.333333
.25
.2
.166666
```

```
READY
@EDI 80?I?I:? "END"
@RUN
1
SN ERROR IN LINE 10
```

```
READY
@LIS80
80NEXTGI:? "END"
```

```
READY
@
```

": " followed by another statement is not allowed to immediately follow the NEXT statement.

```
@EDI80:? "END"
@EDI10,6,6:?I;
@RUN
1 1
2 .5
3 .333333
4 .25
5 .2
6 .166666
```

```
READY
@LIST 10
10 FOR I = 1 TO 6:?I;
```

```
READY
@EDI 80?I?K
@RUN
1 1
NX ERROR IN LINE 80
```

```
READY
@LIS 80
80 NEXT K
```

```
READY
@
```

The Control Variable in the FOR statement is I while that in the NEXT statement is K.

The FOR loop formed by statement 10 is disabled when statement 30 is executed.

[illegible]

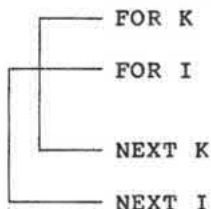
```

READY
@LIST
10 FOR K = 1 TO -6 STEP-1: ?I;
30 FOR I = 7 TO 9 STEP.5
40 PRINT 1/I
60 NEXT K
80 NEXT I

READY
@

```

In this example, there is an illegal nested FOR loop.



## 5.2 Conditional Control Transfer

Conditional statements are used to examine a specific condition during program execution so as to change order of execution as required. The condition in the conditional statement is a relational expression. The truth value of the relational expression is tested so as to determine whether program execution is to be changed.

### 5.2.1 IF...THEN

Format:

If rexpr THEN snum|statement

Description:

When rexpr is true, program execution will go on from the statement following THEN. If a statement number (snum) follows THEN, the program execution will jump to that statement number. When rexpr is false, program execution will go on from the statement immediately following the IF...THEN statement.



Execution mode:

Deferred mode.

As program execution goes on from the statements following THEN when rexr is true, it is possible to find a group of statements separated by ":" put after THEN. However, one should not place ":" statement after THEN snun.

Let us look at the following example:

```
@NEW
@10 INPUT N,M
@20 IF N>M THEN 50
@30 PRINT ,M
@40 END
@50 PRINT N
@60 END
@RUN
?12,49
49

READY
@RUN
?10,5
10

READY
@NEW
@10 INPUT N,M
@20 IF N>M THEN ?N:END
@30 PRINT ,M:END
@RUN
?12,49
49

READY
@RUN
?10,5
10

READY
@NEW
@10 INPUT N
@20 IF N<100 THEN N=N+1:GOTO40
@30 PRINT "N">=100"
```

```
@40 PRINT "N=";N
@RUN
?49
N=50
```

```
READY
@RUN
?120
N>=100
N=120
```

```
READY
@
```

## 5.2.2 More on Loops

The conditional control transfer statement IF and the unconditional control transfer statement GOTO can be paired to form a powerful type of loop.

In this type of loop, there can be more than one control variables, and the creation and modification of the control variables can be designed by the programmer. However, in the FOR/NEXT loops described before, the control variable can only be incremented or decremented as specified by the STEP acxpr-3. Hence, in a strict sense, the FOR/NEXT loop can be considered as a special case of the IF/GOTO loop.

In an IF/GOTO loop, the control variable can be modified through addition, subtraction, multiplication, division or other more complicated operations, (For more detail on the GOTO statement, see 5.3) which can hardly be done with the FOR loop. The GOTO statement is an unconditional control transfer, each time it is implemented the program execution will jump to the statement number specified in the GOTO statement.

For example,

```
@NEW
@10 PRINT = 1 TO 6
@20 PRINT 1/I
@30 NEXT I
```

is the same as the following one:

```
@NEW
@10 I=1
@20 IF I>6 THEN 50
@30 PRINT 1/I
@40 I=I+1:GOTO20
@50 END
@RUN
1
.5
.333333
.25
.2
.166666
```

READY

@

As mentioned earlier, in an IF/GOTO loop, the modification of control variables can be accomplished through addition, subtraction, multiplication, division, or other more complicated operations. See the following examples:

```
@NEW
@10 I=1
@20 IF I>90 THEN 50
@30 PRINT 1/I
@40 I=I*2:GOTO20
@50 END
@RUN
1
.5
.25
.125
.0625
.03125
.015625
READY
@NEW
@10 I=2
@20 IF I>999999 THEN 50
@30 PRINT I
@40 I=I*I:GOTO20
@50 END
```

```
@RUN
2
4
16
256
65536
```

```
READY
@
```

## 5.3 Unconditional Control Transfer

As you know, normally program execution starts with that of the least statement number and go on in the numeric order of statement numbers one after another. In case at certain points of a program, you want to override the normal sequential order so as to transfer control to a certain statement number and go on program execution therefrom, you can use the unconditional control transfer, the GOTO statement.

### 5.3.1 GOTO

GOTO - Transfer of execution order.

Format:

GOTO Snum

Description:

After this statemnt is executed, the next statement to be executed is the statement whose statement number is specified by snum.

Note:

If Snum is not to be found anywhere in the program, the following error message will be displayed:

UL ERROR IN LINE

Remark:

We have described the GOTO command in Chapter 3, from the viewpoint of execution mode, we can say that the GOTO command is of the immediate execution mode while the GOTO statement is of the deferred mode.

When the GOTO statement is executed, the next statement to be executed will be that specified by the statement number snum. Therefore, the GOTO statement should not be followed by the statements group preceded by ":". This is because the statements group put there will never be executed.

Try the following example:

```
@NEW
@10 I = 15
@20 PRINT I/2+3
@30 IF I>40 THEN END
@40 I=I+3:GOTO20:I=0
@RUN
10.5
12
13.5
15
16.5
18
19.5
21
22.5
24
READY
@
```

### 5.3.2 Infinite Loop

In Section 5.1, we have mentioned that the control variables are an indispensable part of a loop for it controls the number of times that a loop is to be executed. In practice, there are infinite loops without control variables either due to the specific requirement of a program or simply because of incorrect programming.

In an infinite loop, there is no TEST/EXIT, owing to the absence of the control variables. Thus, if the program execution happens to fall into the body of the infinite loop, it will cause the loop to execute infinitely because there is no exit. In this case the loop execution will not end until there is an error or an external interrupt.

See the following example:

```
@NEW
@10 I=1
@20 PRINT 1/I
@30 I=I+5:GOTO20
@RUN
```

```
1
.166666
.090909
.0625
.047619
.038461
```

```
.
.
.
```

```
CONTROL C 
```

```
READY
```

```
@NEW
@10 PRINT"HELLO"
@20 GOTO10
@RUN
HELLO
HELLO
HELLO
HELLO
HELLO
```

```
.
.
.
```

```
CONTROL C 
```

```
READY
```

```
@
```

## 5.4 Computed Control Transfer

As mentioned earlier, execution of the unconditional control transfer statement GOTO will transfer control to the statement specified by the statement number snum regardless of the number of times of execution on any occasions.

In this Section, we will discuss the computed GOTO statement which can transfer control to one of several statements specified. The statement to which the control is to be transferred will be determined by the numeric value evaluated from an arithmetic expression in the ON aexpr GOTO Snum statement.

### 5.4.1 ON...GOTO

ON...GOTO... - Computed Control Transfer

Format:

On aexpr GOTO snum {[snum]}

Description:

Suppose the statement number list following GOTO read as snum-1, snum-2,.... snum-n, then if the integer part of aexpr is evaluated to be i, the control of program execution will be transferred to snum-i.

Note:

If the value of aexpr is less than 1 or larger than n (n is the number of snum's following GOTO), the following error message will be displayed:

SN ERROR IN LINE

and execution comes to a stop.

Remark:

As with GOTO, ":" statement group is not allowed to follow the ON/GOTO statement. In practice, ON/GOTO is often used as a select switch.

See the following example:

```
@NEW
@10 A0=0:B0=0:A1=0:B1=
@20 INPUT A,e
@30 ON C GOTO 40,60,80
@40 A0=A0+A:A1=A1+1
@50 GOTO 20
@60 B0=B0+A:B1=B1+1
@70 GOTO 20
@80 PRINT"1GROUP :";
@90 PRINT"SUM=";A0;"AVG";
@95 PRINT"=";A0/A1
@100 ?"2GROUP :SUM=";B0;
@110 ?"AVG=";B0/B1
@RUN
?19,1
?50,2
?17,2
?13,1
?12,1
?45,2
?34,2
?11,1
?79,1
?100,1
?55,3
1GROUP :SUM= 234 AVG 39
2GROUP :SUM= 146 AVG 36.5

READY
@
```

MPF-IP BASIC often uses floating numbers in arithmetic expression. Floating number may be the value of variables, array, constants, expressions, or functions.



---

## Chapter 6

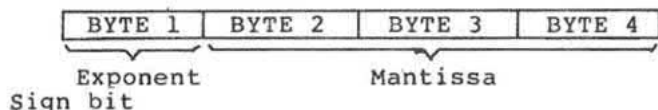
### Numeric Operation

---

In MPF-IP BASIC numeric operation is done in floating number system. Expressed in floating number are the values of variables, arrays, constants, expressions and functions.

#### 6.1 The Notation of Numeric Values in Memory

In MPF-IP BASIC, all numeric values are expressed in floating number notation. In memory, each numeric value takes up four Bytes, i.e., each floating number is expressed in 32 bits. 24 bits are used to express mantissa, 7 bits are for exponent.



```
0=+
1=-
0180000=0.5x2=1
0280000=0.5x4=2
04A0000=(0.5+0.125)x2 =10
80FFFF=-0.999999
```

As all numeric values are expressed in floating number notation, we can not expect absolute accuracy (The deviations are usually negligible though). In operations on some built-in functions, only approximate results can be obtained.

## 6.2 Numeric Functions

The MPF-IP BASIC provides the users with a variety of built-in functions, in this section, you will see some numeric functions.

### 6.2.1 ABS

ABS - Absolute value function

Format:

ABS (aexpr)

Description:

ABS returns the absolute value of the expression.

Note:

In practice, the argument of the ABS function should be a numeric expression, if a string expression is used, the result will be an unpredictable positive number.

```
@?ABS(-10,6)
10,6
@?ABS(A$)
.041962
@?ABS(A$)
.513916
@?ABS(-22E12)
2.19999E+13
```

### 6.2.2 ATN

ATN - Arctangent function

Format:

ATN (aexpr)

Description:

ATN returns the arctangent of an arithmetic expression.

**Note:**

The value of ATN (aexpr) is in radians. If the value in degrees is required, the following equation will do the conversion:

$$\text{Degrees} = \text{Radians} * 180/\text{PI}$$

$$\text{The range of ATN is } -\frac{\text{PI}}{2} < \text{ATN} < \frac{\text{PI}}{2}$$

See the following example:

```
@NEW
@?PI,ATN(1)/PI
3.14159
@?ATN(1)/PI*180
45
@?ATN(0)/PI*180
-2.57839E-04
@?ATN(1.732)/PI*180
59.9993
@
```

### 6.2.3 COS

COS - Cosine function

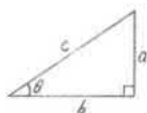
Format:

COS (aexpr)

Description:

COS returns the cosine of an arithmetic expression.

(see Fig. where  $\text{COS}(\theta) = \frac{b}{c}$ )



Note:

aexpr is in radians.

Degrees = Radians \* 180/PI

See the following example:

```
@NEW
@?COS(90)
-.448162
@?COS(PI/2)
-.999997
@?COS(PI*2)
.999997
@?COS(PI/3)
.499999
@
```

## 6.2.4 EXP

EXP - Natural Exponentiation Function

Format:

EXP (aexpr)

Description:

EXP returns  $e^x$ , where  $e$  is the Napierian constant, 2.71827. EXP is the inverse of the natural logarithm function.

See the following example:

```
@?EXP(1)
2.71827
@NEW
@10 FOR I = 2 TO 5
@20 ?EXP(I)
@30 NEXT I
@RUN
7.38905
20.0855
54.5981
148.412
```

```
READY
@?EXP(LN(2))
1.99999
```

## 6.2.5 INT

INT - Integer Function

Format:

INT (aexpr)

Description:

INT returns the largest integer less than or equal to aexpr.

In fact, INT works as a Gauss function.

See the following example:

```
@?INT(3.6)
3
@?INT(-3.6)
-4
@?123.4567E8;INT(123.4567E8)
1.23456E+10 1.23456E+10
@?INT(0,4)
0
@?INT(0,4)
0 -1
```

## 6.2.6 LN

LN - Natural Logarithm Function

Format:

LN (aexpr)

Description:

LN returns the natural logarithm of aexpr. It is the inverse of EXP.

Note:

aexpr must be greater than zero, otherwise the following error message will be displayed:

OF ERROR IN LINE

See the following example:

```
@?LN(0)
OF ERROR IN LINE
@?LN(-12)
OF ERROR IN LINE
@?LN(EXP(5))
  4.99999
@?LN(123456)
  11.7236
@?LN(10)
  2.30250
```

## 6.2.7 LOG

LOG - Common Logarithm Function

Format:

LOG (aexpr)

Description:

LOG returns the common logarithm of aexpr. The LOG log10(aexpr). It is the inverse of 10x function. Note the following relation:

$LOG(X) = LN(X) / LN(10)$

Note:

As with LN function, aexpr must be greater than zero, otherwise the following error message will be displayed.

OF ERROR IN LINE

See the following example:

```
@?LOG(0)
OF ERROR IN LINE
@?LOG(10^8)
7.99999
@?LOG(7)
.845097
@?LN(7)/LN(10)
.845097
@?LOG(1)
0
e
```

## 6.2.8 RND

RND - Random number function

Format:

RND [(aexpr)]

Description:

When omitted, aexpr can be considered as 1. RND returns a random value in the current random number cycle. The value is greater than or equal to zero and less than aexpr. The value of aexpr can be negative. In this case, the resultant value will fall between zero and aexpr.

See the following examples:

```
@NEW
@10 INPUT A
@20 FOR I=1 TO A STEP SGN(A)
@30 PRINT INT(RND(A))
@40 NEXT I
@RUN
?7
4
5
0
1
4
4
```

READY

@RUN

?7

4

5

1

0

1

4

4

READY

@5RANDOM1EE

@RUN

?7

0

1

3

1

0

2

5

READY

@RUN

?7

5

3

6

0

0

6

6

READY

@RUN

?-3

-1

0

0

0

-2

READY

@



## 6.2.9 SGN

SGN - Sign function

Format:

SGN (aexpr)

Description:

SGN (X) = 1 if  $X > 0$

SGN (X) = 0 if  $X = 0$

SGN (X) = -1 if  $X < 0$

See the following two examples, which have the same effect.

```
@NEW
@10 INPUT A
@20 PRINT SGN(A)
@30 GOTO 10
@NEW
@10 INPUT A
@20 IF A<>0 THEN A=A/ABS(A)
@30 PRINT A
@40 GOTO 10
```

## 6.2.10 SIN

SIN - Sine function

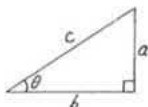
Format:

SIN (aexpr)

Description:

The SIN function returns the sine of an aexpr.

See Fig. where  $\text{SIN}(\theta) = \frac{a}{c}$



Note:

aexpr must be in radians.  
Use the following equation for conversion.  
 $\text{Degrees} = \text{Radians} * 180/\text{PI}$

See the following example:

```
@NEW
@?SIN(90)
.89395
@?SIN(PI/2)
.999997
@?SIN(PI)
1.19209E-07
@?SIN(PI*2)
-2.38418E-07
@?SIN(PI/3)
.866025
@
```

## 6.2.11 SQR

SQR - Square root function

Format:

SQR (aexpr)

Description:

$\text{SQR (aexpr)} = \text{aexpr}(1/2)$

Note:

The value of aexpr must be positive or zero, otherwise the following error message will be displayed:

OF ERROR IN LINE

See the following example:

```
@?SQR(9);9^(1/2)
2.99999 2.99999
@?SQR(7)
2.64575
@?SQR(0)
0
@?SQR(-8)
OF ERROR IN LINE
@?(-8)^.5
OF ERROR IN LINE
@
```

## 6.2.12 TAN

TAN - Tangent function

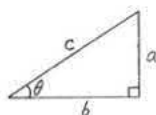
Format:

TAN (aexpr)

Description:

TAN returns the tangent of an aexpr. See Fig. where

$$\text{TAN}(\theta) = \frac{a}{b}$$



Note:

aexpr must be in radians. TAN is the inverse of the ATN function. Use the following equation for conversion.

$$\text{Degrees} = \text{Radians} * 180/\text{PI}$$

See the following example:

```
@NEW
@?TAN(PI/4)
.999999
@?TAN(PI)
-1.19209E-07
@?TAN(45)
1.61988
@?TAN(PI/2)
8.38858E+06
@?TAN(PI/3)
1.73205
@
```

---

## Chapter 7 Array

---

An array is also called a Matrix which is composed of a group of variables with the same name. For identification of each variable, a subscript is added to the common name. For instance,  $H(5)$  represents the fifth (or sixth in a strict sense) element of the array  $H$ .

An array can either be one-dimensional or two-dimensional. A one-dimensional array is composed of a single column with a number of rows. In practice, a subscript is added to specify the row. The numbering of columns and rows starts from zero. A two-dimensional array can be conceptualized as a table with multiple columns and rows. For instance:  $DIM A(3,4)$  can be tabulated as follows:

|      |   | Columns  |          |          |          |          |
|------|---|----------|----------|----------|----------|----------|
|      |   | 0        | 1        | 2        | 3        | 4        |
| ROWS | 0 | $A(0,0)$ | $A(0,1)$ | $A(0,2)$ | $A(0,3)$ | $A(0,4)$ |
|      | 1 | $A(1,0)$ | $A(1,1)$ | $A(1,2)$ | $A(1,3)$ | $A(1,4)$ |
|      | 2 | $A(2,0)$ | $A(2,1)$ | $A(2,2)$ | $A(2,3)$ | $A(2,4)$ |
|      | 3 | $A(3,0)$ | $A(3,1)$ | $A(3,2)$ | $A(3,3)$ | $A(3,4)$ |

In a two-dimensional array, two subscripts separated by a comma are used for each element. As shown in the above table, the first subscript is used to specify the row number, and the second the column number.

In MPF-IP BASIC, there are two methods to define an array.

(1) Explicit Type:

In this method, an array is declared by a DIM statement (see next section). In the statement, the name of the array, the number of rows and the number of columns are specified.

(2) Implicit Type:

It is possible for you to use the elements of an array without declaration by a DIM statement beforehand. In this case, if the array is one-dimensional, the system will automatically set the variable name as that of an array with 11 elements (subscripts range from zero to 10).

If the array is two-dimensional, the system will set the variable name as the name of a two dimensional array with 121 elements (subscripts range from 0 to 10 for both row and column).

In MPF-IP BASIC, you can not use an array for string variables.

## 7.1 DIM

DIM- Declaration of an array.

Format:

DIM array subscript [{,name subscript}]

Description:

DIM is used to declare an array

Note:

The number of subscripts for an array variable can only be one or two, i.e., only one dimensional and two dimensional arrays are allowed in practice.

Remark:

The array is most often used to formulate a table. During the program execution, one can easily find any item on the table with the help of the subscripts.

There is a great variety of applications of tables on the computer. For the simulation of the advanced applications such as stack, queue, order lists, tables together with subscripts are widely utilized and are found powerful.

During the program execution, if the value of a subscript exceeds the range either in explicit or in implicit mode, the following error message will be displayed:

```
SN ERROR IN LINE
```

and the program execution will halt. Try the following examples:

```
@NEW
@DIMZ(20,9)
@Z(10,4)=8:Z(10,4)
8
@?Z(0,0);Z(20,9)
0 0
@? Z(20,10)
SN ERROR IN LINE
READY
@?A(11,11)
SN ERROR IN LINE

READY
@?A(10,10)
0
```

```
@NEW
@10 REM PRIME NUMBER
@20 DIM P(50)
@30 P(0)=2
@40 PRINT 2;
@50 PO=0
@60 P1=3
@70 FOR I=0 TO P0
```

```

@80 2F P(I)>SQR(P1)+1 THEN 120
@90 J=P(I)
@100 IF INT(P1/J)*J=P1 THEN 150
@110 NEXT I
@120 P0=P0+1:P(P0)=P1
@130 IF POS(0)>=20 THEN ?
@140 PRINT P1;
@150 P1=P1+2
@160 GOTO70
@RUN
  2   3   5   7   11  13
17   19  23  29  31
37   41  43  47  53
59   61  67  71  73
79   83  89  97 101
103  107 109 113
127  131 137 139
149  151 157 163
167  173 179 181
191  193 197 199
211  223 227 229
233 SN ERROR IN LINE 120

```

## 7.2 Changing the Dimension of an Array

When the DIM statement for an array is executed, each and every element of the array is given a default value of zero. With the execution of a DIM statement, the Maximum Storage Size for the array variables is fixed, and it is impossible to change the Maximum Storage Size until the execution of RUN or NEW statement. In response to the second, third,...DIM statement for the array variables, the system will check if the Maximum Storage Size of the new DIM statement is greater than that of the first DIM statement. If so, the following error message will be displayed:

```
SN ERROR IN LINE
```

and the program execution will come to a halt. Otherwise, the system will only change the dimension of the array and retain the contents of the storage.



Try the following examples:

```
@NEW
@10 DIM A(10)
@20 FOR I=0 TO 10
@30 ?A(I);:A(I)=I:?A(I)
@40 NEXT I
@50 DIM A(1,4)
@60 FOR I=0 TO 4
@70 FOR J=0 TO 1
@80 PRINT A(J,I)
@90 NEXT J
@100 PRINT
@110 NEXT I
@RUN
```

```
0 0
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0 9
0 10
1 5
1 6
2 7
3 8
4 9
```

```
READY
@DIM A(2,4)
SN ERROR IN LINE
```

```
READY
@
```

In MPF-IP BASIC, the one-dimensional arrays is considered as a special case of the two-dimensional array. For instance, A(I) is considered an equivalent to A(I,0).

Try the following example:

```
@NEW
@10 DIM A(10)
@20 FOR I=0 TO 10
@30 A(I)=I
@40 NEXT I
@50 FOR I=0 TO 10
@60 PRINT A(I);A(I,0)
@70 NEXT I
@RUN
```

|    |    |
|----|----|
| 0  | 0  |
| 1  | 1  |
| 2  | 2  |
| 3  | 3  |
| 4  | 4  |
| 5  | 5  |
| 6  | 6  |
| 7  | 7  |
| 8  | 8  |
| 9  | 9  |
| 10 | 10 |

READY

@

---

## Chapter 8

### String Operation

---

MPF-IP BASIC provides the user with a variety of functions to deal with strings. In operands there are string literals, string variables and string functions. In statements, there are assignment statements, concatenation statements and I/O (input/output) statements.

The main purpose to provide the capability of processing strings is to make MPF-IP a programmable numeric calculator as well as a documentation processor.

#### 8.1 String Literals

A string literal is composed of characters. The number of characters in a string can range from zero to 255. In MPF-IP BASIC, any character corresponds to a number between 0 and 255 according to the ASCII convention. (See Appendix A). We are familiar with some of the characters such as letters of the alphabet, numbers 0 to 9, and special symbols in general use. Others include figures or symbols not given in standard ASCII code. These characters (some are irregular graphics) are peculiar to the MPF-IP system and are entered through the keyboard in combination with the CONTROL key. In practice, all the characters stated above can be used in a string literal.

A string literal is a group of characters enclosed by a pair of quotes or apostrophes as shown below:

```
"CHARACHER STRING"  
'ANOTHER STRING'
```

Examine the following examples:

```
"I'M BASIC"  
"ABC"  
'QUOTE MARK (")'  
'MPF-IP'
```

When quotes or apostrophes are required within a string literal, be sure to use the other as the enclosure as shown above.

## 8.2 String Variable

The string variable is used to store a string of ASCII characters. The length of a string variable can be changed as required, which can be evaluated by the LEN function. The length of a null string is zero.

In theory, the maximum length of a string variable can reach 255. In fact, however, due to the limited size of the memory, there are some restraints.

In execution, a string variable is given a default value of null string by the MPF-IP BASIC.

The name of a string variable is a letter of the alphabet followed by a dollar sign "\$", or a letter followed by any of the numbers from zero to 9 which is again followed by a dollar sign.

In practical application of string variables, the name of the string variable is used to represent the whole string variable. To use a substring, however, string functions such as MID\$, LEFT\$, RIGHT\$ are required. In MPF-IP BASIC, the notation of pseudo variable is not accepted, thus it is more or less inconvenient to change a substring. Notwithstanding, it is still possible to change some substrings of a string with the help of string functions or string expressions.

See the following examples:

```
@A$="THIS IS AN EXAMPLE"
@?A$
THIS IS AN EXAMPLE
@?LEFT$(A$,7)
THIS IS
@?MID$(A$,8,3,)
AN
@?RIGHT$(A$,11)
EXAMPLE
@?A$="THESE ARE"+RIGHT$(A$,11)
@?A$
THESE ARE EXAMPLE
@A$=A$+"S"
@?A$
THESE ARE EXAMPLES
```

### 8.3 String Expression

An string expression is composed of a single string (including string literal, string variable and string function) or two strings combined by a concatenation operator "+". The concatenation operator functions to combine the two strings in the order from left to right to form a new string. A string expression is used to be assigned to a string variable or to form a relational expression (For more detail refer to 8.5 on the comparison of strings).

The format of string expression is as follows:

string

string + string

Here the string can be a string literal, a string variable, a string variable or a string function. In application, it is a good practice to avoid combining many strings at a time as shown below in order to avoid undesirable results.

string + string + string .....

Examples of string literal, string variable, and string function are "BASIC", "MPF-IP", A\$, Z9\$, and CHR\$(65), Num\$(100) respectively.

The length of an evaluated string expression is also restricted to zero to 255.

See the following examples:

```
@? "ABC"+" COMPUTER"
ABC COMPUTER
@C$="BASIC"
@?C$+"-MPF-IP"
BASIC-MPF-IP
@?CHR$(90)+C$
ZBASIC
@?CHR$(66) "ASIC"
BASIC
@CHR$(65)+CHR$(90)
AZ
@A$=" COMPUTER"
@?C$+A$
BASIC COMPUTER
@
```

## 8.4 Functions for Strings

A number of built-in functions for strings are available on MPF-IP BASIC. Among them, some are used for conversion between numeric values and strings. In this section, we present the functions in alphabetical order. Below is a list of them in four categories:

(1) For the operation of substrings:

|         |         |
|---------|---------|
| LEFT\$  | (8,4,4) |
| MID\$   | (8,4,6) |
| RIGHT\$ | (8,4,8) |

(2) For the formation of new strings:

|          |          |
|----------|----------|
| SPACE\$  | (8,4,9)  |
| STRING\$ | (8,4,10) |

(3) For the conversion between numeric values and strings:

|       |        |
|-------|--------|
| ASCII | 8,4,1  |
| CHR\$ | 8,4,2  |
| NUM\$ | 8,4,7  |
| VAL   | 8,4,11 |

(4) Others

|       |       |
|-------|-------|
| INSTR | 8,4,3 |
| LEN   | 8,4,5 |

### 8.4.1 ASCII

ASCII - ASCII code function

Format:

ASCII (sexpr)

Description:

ASCII (American Standard Code for Information Interchange) is a set of codes devised by the American National Standards Institute for effective interchange of information between different computers.

Every character ( letter of alphabet, number, or symbol) is given a corresponding ASCII code. According to its definition, every ASCII code is composed of 7 bits ranging from (0000000) to (1111111) in binary representation or zero to 127 in decimal.

The ASCII function returns the corresponding ASCII value of the first character of the string expression sexpr. (For the table of ASCII values, please refer to Appendix A.)

Remark:

We have mentioned in the above that each standard ASCII code is composed of 7 bits. As the CPU (Central Processing Unit) of MPF-IP processes the data on a base of 8 bits, it is convention to regard the most significant bit (MSB) as zero. The CHR\$ function is in a sense the inverse of the ASCII function, in the discussion on CHR\$, we will examine the problem of correspondence when a numeric value is larger than 2<sup>18</sup> (=256).

See the following examples:

```
@?ASC("0");ASC("01")
48 48
@?ASC("1");ASC("9")
49 57
@?ASC("AX");ASC("ZZ")
65 90
@NEW
@10 INPUT K$
@20 PRINT ASCII(K$)
@30 GOTO 10
@RUN
?UJG
85
?+++
43
?" "
0
?<
60
?"GG
CV ERROR IN LINE 10

READY
@RUN
?"AAA"
65
?" "
34
?" "
39
?"FF"
CV ERROR IN LINE 10
READY
@
```



In the above examples, the error message CV ERROR occurred twice because the input string did not properly adhere to the "" or "" rules. For the input of a string, the use of "or" will make no difference provided it is in legal form. (For more detail on the input of strings, please refer to chapter 9.)

## 8.4.2 CHR\$

CHR\$ - Character function

Format:

CHR\$ (aexpr)

Description:

CHR\$ is a string function which returns a one-character string which contains the alphanumeric equivalent of the argument, according to the conversion table in Appendix A.

The ASCII code is formed by 7 bits while Z-80A is based on 8 bits. Consequently, for any value of the numeric expression aexpr between zero and 255, the corresponding character will be displayed according to the ASCII conversion. In case aexpr is greater than 127, the value will be subtracted by 128. Displayed on the printer will be values between zero and 127. If aexpr exceeds 255, unpredictable characters will be displayed.

Try the following examples:

```
@NEW
@10 INPUT A
@20 PRINT CHR$(A)
@30 GOTO 10
@RUN
765
A
?34
"
```

```

?78
N
?200
H
?193
A
? CONTROL C
STOP AT LINE 10

READY
@?ASC(CHRS(65))
65

@?ASC(CHRS(1222))
4
@?CHRS(ASC("ABC"))
A

```

### 8.4.3 INSTR

INSTR - The position of a string in another string.

Format:

```
INSTR(aexpr, sexpr-1, sexpr-2)
```

Description:

The INSTR function is used to find the position of sexpr-2 in sexpr-1, where the starting position for search and comparison is controlled by aexpr which must be a positive number.

INSTR Returns:

- (1) 0 - when sexpr-2 is not to be found in sexpr-1 after the aexpr-th character.
- (2) 1 - when sexpr-2 is a null string
- (3) n - when sexpr-2 is found starting from the n-th character in sexpr-1.

Try the following example:

```
@NEW
@5 ?"INPUT MAIN STRING"
@10 INPUT S1$
@15 ?"INPUT SUBSTRING"
@20 INPUT S2$
@25 ?"INPUT STARTING POINT"
@30 INPUT A
@40 ? INSTR(A,S1$,S2$)
@45 ?"INPUT SELECTION"
@50 INPUT K
@60 ON K GOTO 5,15,25
@RUN
INPUT MAIN STRING
?ABCDEFGHJKLMNO
INPUT SUBSTRING
?EFG
INPUT STARTING POINT
?1
5
INPUT SELECTION
?3
INPUT STARTING POINT
?6
0
INPUT SELECTION
?2
INPUT SUBSTRING
?2
INPUT SUBSTRING
?"
INPUT STARTING POINT
?1
1
INPUT SELECTION
?2
INPUT SUBSTRING
?ADDGY
INPUT STARTING POINT
?1
0
INPUT SELECTION
?1
INPUT MAIN STRING
?ADDCGADDGY
```

```

INPUT SUBSTRING
?ADD
INPUT STARTING POINT
?1
1
INPUT SELECTION
?3
INPUT STARTING POINT
?3
6
?CONTROL C
STOP AT LINE 50

READY
@

```

#### 8.4.4 LEFT\$

LEFT\$ - Left substring function

Format:

```
LEFT$(sexpr, aexpr)
```

Description:

LEFT\$ returns a string composed of the leftmost aexpr characters of sexpr. When aexpr is larger than the length of sexpr, the LEFT\$ function returns the whole of sexpr. If aexpr is negative, the following error message will display:

```
OV ERROR IN LINE
```

Try the following example:

```

@NEW
@10 INPUT S$
@20 FOR I=1 TO 15
@30 PRINT LEFT$(S$,I)
@40 NEXT I
@RUN
?CHENG-YIH-HWA

```

```

C
CH
CHE
CHEN
CHENG
CHENG-
CHENG-Y
CHENG-YI
CHENG-YIH
CHENG-YIH-
CHENG-YIH-H
CHENG-YIH-HW
CHENG-YIH-HWA
CHENG-YIH-HWA
CHENG-YIH-HWA

```

```

READY
@?LEFT$(S$, -1)
OV ERROR IN LINE

```

As you will find in 8.4.6 on the MID\$ function, LEFT\$ can be expressed by MID\$ as shown in the following equation.

$$\text{LEFT}$(aexpr, aexpr) = \text{MID}$(sexpr, 1, aexpr)$$

And in a later section you will find

$$\text{sexpr} = \text{LEFT}$(sexpr, aexpr) + \text{RIGHT}$(sexpr, aexpr+1)$$

### 8.4.5 LEN

LEN - String length function

Format:

LEN (sexpr)

Description:

The LEN function returns an integer equal to the number of characters in the string argument. The value may range from zero to 255, i.e., a string can contain 255 characters at most.

Try the following example:

```
@NEW
@10 INPUT S$
@20 FOR I=1 TO LEN(S$)
@30 PRINT LEFT$ (S$,I)
@40 NEXT I
@RUN
?BASIC

B
BA.
BAS
BASI
BASIC

READY
@
```

#### 8.4.6 MID\$

MID\$ - Middle part of a string

Format:

MID\$ (sexpr, aexpr-1, aexpr-2)

Description:

The MID\$ function returns a substring of sexpr composed of aexpr-2 characters starting from the (aexpr-1)th character.

Remark:

Be sure you understand the following equations:

(1) MID\$ (sexpr, 1, N) = LEFT\$ (sexpr, N)

(2) MID\$ (sexpr, N, LEN(sexpr)-N+1)=RIGHT\$(sexpr,N)

For more detail on the RIGHT\$ function, please see 8.4.8.

```

@NEW
@10 INPUT S$
@20 FOR I=1 TO LEN(S$)
@30 A$=MID$(S$,I,1)
@40 ?A;ASCII(A$)
@50 NEXT I
@RUN
?BASIC-MPF-IP
B 66
A 65
S 83
I 73
C 67
- 45
M 77
P 80
F 70
- 45
I 73
P 80

REALY
@?MID$(S$,6,5)
-MPF-
@?LEFT$(S$,5)
BASIC
@?MID$(S$,1,5)
BASIC
@

```

#### 8.4.7 NUM\$

NUM\$ - Conversion from a number to a string

Format:

NUM\$(aexpr)

Description:

The NUM\$ converts the resultant value of the numeric expression aexpr to a string representation.

Note:

If a string expression is entered as the argument of the NUM\$, the NUM\$ will return zero.

## Description:

The usage of the NUM\$ function can be considered as a special method of input/output. In fact, it is called an Internal Output Function while the usage of PRINT statement is usually referred to as an External Output. In 8.4.11 on VAL function, we will examine the relation between Internal and External Input.

In fact, there is a conversion step contingent to all output execution. When we use the PRINT statement, a numeric expression is first evaluated and then the result is displayed on the screen or printed on a printer. When the NUM\$ function is used, however, the result is stored at a specified address in the memory. That is, the programmer can reuse the result of the conversion as a string or assign and store the string to a string variable. Similarly, the resultant string can be used to form another string expression or used in the comparison of strings. In short, the result of conversion can be reused in the case of NUM\$ while the result can only be output to a display screen or printer in the case of the PRINT statement.

Note the following equation:

```
PRINT aexpr = PRINT NUM$ (aexpr)
```

The NUM\$ function is the inverse of the VAL function which is described in 8.4.11.

```
@?5;"AAA"  
5 AAA  
@?NUM$(5);"AAA"  
5 AAA  
@NEW  
@10 INPUT A,B  
@20 A$=NUM$(A)  
@25 IFA$=" 0 " THEN END  
@30 B$=NUM$(B)  
@40 S$=NUM$(A+B)  
@50 L$=A$+"+":L1$=B$+"=" "  
@60 L$=L$+L1$:L$=L$+S$  
@70 PRINT L$  
@80 GOTO10
```



```

@RUN
?12,78
  12 + 78 = 90
?5E7, 6E6
  5.000000E+07+6.000000E+06=5.600000E+07
?123.5,-67.3
  123.5+-67.3=56.2
?1.0,10000000
  1+1,0000000E+06=1.000000E+06
?1E-10,9E-10
  9.999999E-11+8.999999E-10=9.999999E-10
?0,0

```

READY

@

### 8.4.8 RIGHT\$

RIGHT - Rightside substring function

Format:

RIGHT\$(sexpr, aexpr)

Description:

RIGHT\$ returns a substring which includes the aexprth character to the end of sexpr. When the value of aexpr turns out greater than the length of sexpr, RIGHT\$ returns the whole string. If aexpr is negative, the following error message will be displayed:

OV ERROR IN LINE

In 8.4.6, we have described the following relations among MID\$, LEFT\$ and RIGHT\$, test the following examples to see if the relations are true.

RIGHT\$(sexpr, N)=MID\$(sexpr, N, LEN(sexpr)-N+1)  
sexpr=LEFT\$(sexpr, N)+RIGHT\$(sexpr, N+1)

#### Examples:

```
@NEW
@A$="MPF-I-PLUS"
@?RIGHT$(A$,7)
PLUS
@?MID$(A$,7,LEN(A$)-7+1)
PLUS
@?LEFT$(A$,7)+RIGHT$(A$,18)
MPF-I-PLUS
@NEW
@10 A$="MPF-I-PLUS"
@20 FOR I=1 TO LEN(A$)
@30 PRINT RIGHT$(A$,I)
@40 NEXTI
@RUN
MPF-I-PLUS
PF-I-PLUS
F-I-PLUS
-I-PLUS
I-PLUS
-PLUS
PLUS
LUS
US
S

READY
@
```

#### 8.4.9 SPACE\$

SPACE - Space function

Format:

SPACE\$ (aexpr)

Description:

SPACE\$ returns a number of continuous blanks.



```

      **
    *  *
  *      *
 *        *
*          *
 *        *
  *      *
    *  *
      **

```

```

READY
@RUN
?1

```

```

      **
    *  *
  *      *
 *        *
*          *
 *        *
  *      *
    *  *
      **

```

```

READY
@

```

#### 8.4.10 STRING\$

STRING\$ - String of identical characters

Format:

```
STRING$ (aexpr-1, aexpr-2)
```

Description:

STRING\$ returns a string of aexpr-1 identical characters with an ASCII code of aexpr-2.

Remark:

As described in 1.11, you can use the STRING\$ function in its abbreviated form, i.e.,

```
STRING$ = STR(aexpr)
```

As a result, whenever you want to save memory, you can use the STR function instead of a string of characters provided the number of the reiterated characters is greater than that of the bytes occupied by STR(aexpr).



READY

@

#### 8.4.11 VAL

VAL - Value of numeric string function

Format:

VAL (sexpr)

Description:

The VAL function returns the value of a string in the form of numbers.

Note:

If there is any illegal character in the argument, the following error message will be displayed as in the case of the execution of the INPUT STATEMENT.:

CV ERROR IN LINE

Remark:

VAL can be considered as the inverse of NUM\$. It is regarded as a special I/O format i.e., an Internal Input Function. In comparison, the INPUT statement commonly used is called an External Input Function. Earlier in 8.4.7, we have described the relationship between Internal/External Output.

In all forms of input, there is always a conversion involved. This is more important with the input of a string. As mentioned in Chapter 6, in MPF-IP BASIC, a numeric value is stored in memory as a four byte floating point number. What is entered through the keyboard is nothing but individual ASCII codes. Therefore, the process of conversion carries much significance.

Later in chapter 9, we will describe the READ statement. It is a form of internal input just like the VAL function.

In comparison, the INPUT statement converts the ASCII codes entered through the keyboard into numeric values, while the VAL function treats a string in the program as an entry through keyboard and operate on it just like an INPUT statement, and the READ statement fetch a data from a Data Buffer in the program and then operate on it in a way similar to INPUT and VAL. To sum up, we can consider the VAL function as an independent function, while the execution of INPUT and READ request that the data be placed at a definite address and then process of the VAL function is implemented.

In the following example, we will show you how to use the VAL function, and a program is used to convert a positive integer.

```

@NEW
@?VAL("1E7)
1.000000E+07
@?VAL("9E-7")
8.999999E-07
@?VAL(" 123")
123
@10 INPUT N$
@20 N=0
@30 FOR I=1 TO LEN(N$)
@40 N1$=MID$(N$,I,1)
@50 IF N1$=" " THEN 9
@60 IF N1$>"9" THEN 120
@70 IF N1$<"0" THEN 120
@80 N=N*10+ASC(N1$)-48
@90 NEXT I
@100 PRINT N$;"=";N
@110 GOTO 10
@120 PRINT N$
@130 PRINT$PA(I-1);"*CONVERSION ERROR"
@140 GOTO 10
@RUN
?" 45"
45= 45
?67
67= 67
?76543
76543= 76543
?"444 23"
444 23= 44423

```

```

?123L9
123L9
  *CONVERSION ERROR
?1#903
1#903
  *CONVERSION ERROR
?RRR
RRR
  *CONVERSION ERROR
?"4      6 5 "
4      6 5= 465
?"4      $ 677"
4      $ 677
  *CONVERSION ERROR
?|C
STOP AT LINE 10

READY
@

```

## 8.5 Comparing Strings

Comparisons of strings are possible with the relational operators. The result is a truth value which is often used in an IF statement (For more detail, please see ). Listed below are the relational operators:

| <u>Operator</u> | <u>Meaning</u>           |
|-----------------|--------------------------|
| =               | equal to                 |
| <>              | not equal to             |
| <               | less than                |
| >               | greater than             |
| <=              | less than or equal to    |
| >=              | greater than or equal to |



Two strings are equal only if they have the same logical length and each character matches. A string is less than another if its first character that does not match the other is numerically less than (according to the standard ASCII codes for characters) or it is an initial proper subset of the other.

The detailed description of the other relational operators are omitted since they can be easily figured out from the above discussion on "=" and "<".

Try the following example:

```
@NEW
@10 INPUT A$
@20 INPUT B$
@30 PRINT
@40 PRINT A$;
@50 IF A$=B$ THEN ?"=";
:GOTO 80
@60 IF A$<B$ THEN ?" ";
:GOTO 80
@70 PRINT ">";
@80 PRINT B$
@90 GOTO 10
@RUN
?ABC
?ABB

ABC>ABB
?X
?XX

X<XX
?
?" "
?G

<G
?AAAA
?AAAA

AAAA=AAAA
?ASRF
?ASR
```

```
ASRF>ASR
?56
?57
56<57
?186
?19

186<19
?|C
STOP AT LINE 10

READY
@
```

## 8.6 Input/Output of Strings

Detailed description on input/output of strings will be presented in Chapter 9. Topics on PRINT, INPUT, and READ/DATA/RESTORE statements will be found in that chapter. Special emphasis must be given to the "," and ";" in the PRINT statement.

---

## Chapter 9

### I/O Statement

---

I/O plays an essential part in the operation of a computer. In MPF-IP BASIC, the fundamental I/O statements are POS and INP functions and the OUT statement.

Information processing is undoubtedly the major function of a computer system. In preceding chapters we have described the functions of MPF-IP BASIC in connection with the operation on numbers and strings. A computer system has to communicate with the outside world, i.e., the information to be processed must be input onto the computer system, and then the processed information is expected to be output to the outside world. The SAVE and LOAD commands on MPF-IP BASIC are used together with a tape-recorder. The LOAD command will cause the information stored on the magnetic tape transferred to the computer. The LIST command will have the program displayed on a screen or printed on a printer. The SAVE command will have the program entered through the keyboard transferred to and stored on a secondary storage such as a magnetic tape.

For a computer system, there are a number of peripheral devices in the outside world. For MPF-IP, we have the keyboard, magnetic tapes, the printer. In future, the peripheral devices will include a video output and disk drive. In each case, different methods of I/O operations are required for different peripheral devices.

To make the most of a computer system, the user must communicate effectively with it. One can enter the programs and data by the different methods of communication into the computer and get the results. In an application system, the user can do without the I/O operations. Certain devices will automatically provide the system with necessary data, and the output of the system will instruct the peripheral devices to do the subsequent procedures.

On a general-purpose computer system, the user communicates directly with the computer through the I/O operations which are the principal topics of this chapter. As described in the section on permanent storage commands, there are devices called auxiliary storage which the computer can access directly.

## 9.1 PRINT

The PRINT statement is used to have some values displayed on the indicator panel. If a printer is attached to the PRT-MPF-IP system, it can be printed on the printer. To know how the printer operates, please see the PRT-MPF-IP Printer Operation Manual. Before the system enters the BASIC mode, you can press CONTROL P to power ON/OFF the printer. In practice, when the printer is active, CONTROL P will turn it off, otherwise, it is turned on.

PRINT - Display on the indicator panel.

Format:

```
PRINT [{expr}[,|;[{expr}]]][,|;]  
? [{expr}[,|;[{expr}]]][,|;]
```

Description:

"?" can be used in place of PRINT.

The PRINT statement is used to output the values of expressions to the indicator panel or the printer.

Execution mode:

Immediate & deferred mode

Remark:

In practice, there are several fundamental types of the PRINT statement as shown below:

- (1) PRINT
- (2) PRINT expr
- (3) PRINT expr, expr
- (4) PRINT expr; expr
- (5) PRINT expr,
- (6) PRINT expr;

Before we examine the above types in detail, we would like to give a few words concerning the concept of a LINE.

In hardware structure, MPF-IP is equipped with a 20-character green Fluorescent Indicator Panel. With the software control, the buffer of a display line can contain up to 60 characters. In other words, in MPF-IP there are a maximum of 60 characters in a line. During the program execution when the output requires a linefeed, the program execution will come to a halt and it will not resume until the user presses  and all characters on the indicator panel are cleared off. In contrast, suppose a printer is installed to MPF-IP and the printer is in active state. Then when the data displayed on the indicator panel requires a linefeed, the system will instruct the printer to get a hardcopy of that line, get a linefeed automatically and the program execution will go on. For this reason, we recommend that the user of MPF-IP have a printer installed so as to enhance the performance of the system.

In type (1) through (4), neither "," nor ";" is present at the end of these statements. When their execution is accomplished, a linefeed is invoked. We will give you a detailed description on the usage of "," and ";" in 9.1.3 and 9.1.4.

### 9.1.1 Output of Numeric Date

In a PRINT statement, except for some control factors (See 9.1.3, 9.1.4, 9.1.5, 9.1.6, 9.1.7), numeric data and string data (See 9.1.2) are the principal components for output. Numeric data can be categorized into (1) numeric constants (2) numeric variables (3) numeric functions (Including built-in functions and user-defined functions). In spite of the different categories, the final result in practice is a numeric value in any case. The PRINT statement is used to display the value on the indicator panel in a certain format.

In 8.4.7 on the NUM\$ function, we have described the output format of numeric values. In fact, the NUM\$ function can be considered as a conversion routine which can convert a numeric value into a string in a specified format and then the string is displayed on the indicator panel. To understand the conversion format, try the following examples.

```
@?7
7
@?123.4567
123.456
@?+2345.789
2345.78
@?-78.0
-78
@?+34.01
34.01
@?789789789
7.89789E+08
@?-3645.7653
-3645.76
@?1E6
1.000000E+06
@?1E5
1.000000
@?1E-5
1.000000E-05
@?1E-4
9.99999E-05
@?1E18
9.99999E+17
```

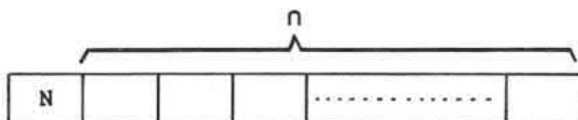
```

@?-1E10
-1.000000E+10
@?-1.E19
OV ERROR IN LINE
4.999999E+18
@

```

## 9.1.2 Output of String Data

String data can be classified into string literal, string variable and string function (including built-in function and user-defined function). Upon the execution of a PRINT statement, the structure of a string data can be conceptualized as follows:



Where each  stands for a byte in the memory. Stored in the first byte is the length of the string (N), followed by n bytes in each is the code for a character of the string.

During the output of string data, if the length of the string is zero, the output pointer will not move. (The length of a null string is zero.)

Try the following example:

```

@?"1273"
1273
@?"'"
'
@?"'"
"
@?""
@?"$FVGR"
$FVGR
@

```

### 9.1.3 The Usage of "," in a PRINT Statement

In MPF-IP BASIC, the maximum length allowed for a line is 32 characters, and every 8 characters form a field. In connection with output, there is an important indicator-the output pointer which points to the next output pointer is "^" (called cursor). In a PRINT statement, the execution of "," will cause the output pointer to jump to the beginning of next field and the subsequent output will start from there. In the usage of ",", when output requires a linefeed, in other words, when the output pointer is in the last field of a line, the subsequent appearance of "," will cause an automatic linefeed, and the output pointer will jump to the beginning of the first field of the next line.

```
10 NEW
20 INPUT J
30 FOR I=1 TO 8
40 PRINT I+J,
50 NEXT I
@ RUN
?0
  1          2          3
    4
  5          6          7
    8
READY
@RUN
?10
  11          12          13
    14
  15          16          17
    18
READY
@
```

If the ";" or "," symbol is embedded in a PRINT command, the BASIC Interpreter will send a linefeed code to the PRT-MPF-IP when the number of characters to be printed out exceeds 32.



#### 9.1.4 The Usage of ";" in a PRINT Statement

Upon the execution of a PRINT statement, both numeric data and string data are converted into the same format which can be considered as string format before they are output. In 9.1.3, we have described that "," is used to control field. In contrast, ";" cause the data to be output one immediately after another after the various data is converted into the same format.

[illegible]

### 9.1.5 Omission of “;”

In MPF-IP BASIC, data is classified into numeric data and string data, which are in turn classified into numeric constants, string constants, numeric variables, string variables, numeric functions, string functions, numeric expressions and string expressions. As mentioned earlier, a variable name begins with a alphabetical letter which may be followed by a numeral. The dollar sign "\$" is added to a string variable. The first two characters of all reserved words used by the MPF-IP BASIC are restricted to letters of the alphabet, which leaves out any possibility of ambiguity in identification. As a result, for a PRINT statement, on condition that the output of data is clearly identifiable, "," may be omitted.

```

@?"1234""5678"
12345678
@?1234"5678"
 1234 5678
@A=10
@?1234A
 1234 10
@?A"5678"
 10 5678

```

In fact, omission of "," is a special feature of MPF-IP BASIC, it is not necessarily allowed in other BASIC.

### 9.1.6 POS Function

POS - Position of output pointer

Format:

POS (0)

Description:

The POS function is used to indicate the current position of output pointer in the output buffer or in a line. You may use this function to prevent the usage of ";" from invoking overflow which destroys other memory contents in the system.

```

@NEW
@10 PRINT
@20 INPUT J
@30 FOR I=1 TO 20
@40 IFPOS(0)>=20 THEN ?
@50 PRINT I+J;
@60 NEXT I
@RUN

```

```

?0
 1 2 3 4 5 6 7
 8 9 10 11 12 13
14 15 16 17 18
19 20
READY
@RUN

```

```
?100
101 102 103 104
105 106 107 108
109 110 111 112
113 114 115 116
117 118 119 120
```

```
READY
@RUN
```

```
?10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
26 27 28 29 30
```

```
READY
@
```

### 9.1.7 TAB Function

TAB - Designation of output position

Format:

TAB (aexpr)

Description:

The TAB function is used together with a PRINT statement to specify the start position of the next output.

Remark:

The TAB function may be considered as a generalized usage of ", ". It is used to specify the start point of the next field in a line. In comparison with ", ", the length of a field is not restricted to 8 (or 4 for the last one) if the TAB function is used.

In practice, the TAB function moves the output pointer to the position specified by aexpr. As a result, if the value of aexpr is less than the current position of the output pointer in the line, a linefeed will occur.

Try the following examples:

```
@NEW
@10 FOR I = 1 TO 10
@20 ?TAB(I*3);I;
@30 NEXT I
  1 2 3 4 5 6
  7 8 9 10
```

```
READY
@ NEW
@10 FOR I=1 TO 10
@20 ? TAB(I);I
@30 NEXT I
@RUN
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
READY
@
```

## 9.2 INPUT Statement

The INPUT statement allows the user to enter through the keyboard the required data during the program execution. In other words, the INPUT statement enables a program to get data from the outside world during the program execution. In addition, the programmer can print a "prompt statement" before the user is requested by the INPUT statement to enter data, this enables the user to acknowledge what he is expected to key in.

INPUT - Input data through the keyboard

Format:

```
INPUT [string,] var [{,var}]
```

### Description:

The INPUT statement enables the programmer to enter data through the keyboard during the program execution.

### Execution mode:

Immediate and deferred mode

### Remark:

The fundamental types derived from the format are listed below:

- (1) INPUT string, var
- (2) INPUT var
- (3) INPUT var, var
- (4) INPUT string, var, var

In (1) and (4), the string will first be printed upon the execution of program and then the system waits for input. In fact they are equivalent to the following statements.

- (1') PRINT string;: INPUT var
- (4') ?string ;: INPUT var, var

As shown in (3) and (4), when two or more items of data are to be input, "," can be used as a delimiter.

If the type of the entered data does not match that of var, the following error message will be displayed:

CV ERROR IN LINE

### 9.2.1 Input of Numeric Data

We have described the conversion of numeric data under the topic on the VAL function in 8.4.11. When you enter a numeric data through the keyboard, the entered item, which is in the form of a string, is first converted into a numeric data through the conversion of the VAL function and then stored onto the specified numeric variable.

```
@INPUT"NUMBER",A:?A
NUMBER?-100
-100
@INPUT C,D
20,45
@PRINT C,D;C;D
20      45 20 45
@NEW:INPUT A,C:?A,C
12,50
12
@INPUT''',A
"?30
@
```

### 9.2.2 Input of String Data

The format of string data input is on the whole the same to that of string literal. Simplified format is allowed as shown in the following examples:

```
@INPUT"STRING",A$:?A$
STRING?"QUIET"
QUIET
@INPUT A$:?A$
?DDD
DDD
@INPUT"TWO",A$,B$:?A$,B$
TWO?"GGG",HHH
GGG      HHH
@INP A$,B$:?A$,B$
?JJJ,GGG
JJJ      GGG
@INP A$,B$:?A$,B$
?"JJJ,GG",HH
JJJ,GG  HH
```

## 9.3 DATA/READ/RESTORE

DATA/READ/RESTORE - Stock of constants in a program

Constants are required in almost all programs. The attempt to relentlessly use the LET statement will make the program over-sized and a lot of memory spaces wasted. In practical applications such as those for industrial and commercial purposes, a set of numeric data or alphanumeric data are required.

In MPF-IP BASIC, the statements DATA/READ/RESTORE are provided to deal with these constant values. We will start with DATA/READ.

In any program, the statements DATA and READ work as a complimentary pair. Either of them can be placed anywhere in a program. DATA and REM are similar in nature, i.e., both of them are non-executable statements. They are different in that anything following REM is ignored by BASIC while those appeared in DATA have actual significance. In fact, the order of statement number of each DATA statement is crucial in program execution. All the DATA statements in a program can be considered as a single DATA statement with the contents of each statement combined in the order of statement numbers.

Upon the execution of a READ statement, there is a conceptualized Data Pointer in the DATA statement pointing to the individual items. The DATA pointer will specify the next item to be read. Each time an item in the DATA statements is read, the data pointer will move to the next one.

The RESTORE statement is used to bring the data pointer back to the first item in DATA statement.

READ - Read an item from DATA

Format:

```
READ var [{,var}]
```

### Description:

The execution of the READ statement will get an item from the DATA statements.

### Execution mode:

Immediate and deferred mode

### Note:

If the number of remaining items in the DATA statements is less than that requested by the READ statement, the following error message will be displayed:

DA ERROR IN LINE

### Remark:

In practice, the numeric constants and string constants can be intermingled in one DATA statement. The data conversion related with DATA is the same as that of INPUT.

DATA - Stock of data

### Format:

DATA string | number [{,string|number}]

### Description:

The DATA statement is used to reserve numeric and/or string constants to be used later in the program.

### Execution mode:

Deferred mode

### Remark:

At the time of programming, all data in the DATA statements are considered as string constants prior to conversion operation. Upon the execution of the READ statement, each item in DATA is converted into a numeric constant or string constant as required.



In the execution of the READ statement, if a non-numeric data is assigned to a numeric variable in the READ statements, the following error message will be displayed..

#### CV ERROR IN LINE

RESTORE - Reset the data pointer

Format:

RESTORE

Description:

The execution of the RESTORE statement will move the data pointer to the first item in the first DATA statement.

Execution mode:

Immediate and deferred mode.

### 9.3.1 Examples

```
@NEW
@10 DATA 1,2,3,4,5,6
@20 DATA 7,8,9,10
@30 FOR I = 1 TO 10
@40 READ J
@50 PRINT J;
@60 NEXT I
@RUN
  1  2  3  4  5  6  7
  8  9 10
READY
@GOTO 30
DA ERROR IN LINE 40
```

```
READY
@RESTORE
@GOTO 30
  1  2  3  4  5  6  7
  8  9 10
READY
@
```

```

@NEW
@10 DATA 1,2,3,4,5,6
@20 DATA 7,8,9,10
@30 FOR I = 1 TO 10
@40 READ JS
@50 PRINT JS;
@60 NEXT I
@RUN
12345678910
READY
@35 RESTORE
@RUN
1111111111
@35 DATA A,B,C,D
@45 READ K
@55 PRINT K;
@RUN
1 2 3 4 5 6 7 8 9 10
CV ERROR IN LINE 45

```

## 9.4 INP Function

INP - Read a data from input port

Format:

INP (aexpr)

Description:

The INP function is used to read a data from input port aexpr, where the range for the data is from zero to 255.

## 9.5 OUT Statement

OUT - Send a data to output port

Format:

OUT aexpr-1, aexpr-2

### Description:

The OUT statement is used to send aexpr-2 to output port aexpr-1, where the range for aexpr-2 is preferably zero to 255.

### Execution mode:

Immediate and deferred mode.

---

## Chapter 10 Subprogram

---

Earlier in the topics on loops, we have mentioned that the main purpose of programming is to make the infinite result out of the limited resources. Like the loops, the subprograms are used to save memory spaces. The subprograms, however, have other valuable features. They greatly increases the readability of a program, and is an excellent tool for structured programming. Moreover, they greatly decrease the difficulty in the development and maintenance of programs.

A subprogram is a predesigned portion of a program which can be used to deal with programs in a specified procedure. Usually a subprogram can not execute independently, it has to be driven by a main program or another subprogram.

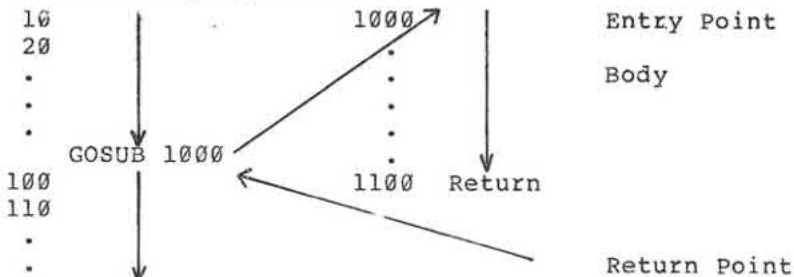
In fact, we have utilized some subprograms in the earlier pages. However, they are called built-in functions then. Examples are numeric value-related functions such as SIN, COS...as well as string-related functions such as MID\$, RIGHT\$...etc. In other words, functions are subprograms in a special form. Each of them returns a value through the operation on a parameter. A subprogram, however, is used to carry out a specified operation, it is used in varied features, not merely for obtaining a value. A special feature of the subprogram makes it possible for us to use a subprogram written in a language other than BASIC. For more details in this respect, please refer to chapter 12.

The modern philosophy of programming advocates modular programming and structural programming. These ideas in programming are based on the notion of Independency derived in the construction of subprograms.

On the basis of Independency, when we set out on programming we start with the abstraction of a problem and then divide the problem into a number of Black Boxes to be processed. Subsequently we make a subprogram out of each black box and then combine them with each one placed in its proper location. The procedure stated above is called modular programming.

## 10.1 Components of a Subprogram

Flow of subprogram execution



The components of a subprogram are described below:

### (1) Entry Point:

The BASIC language is so devised that any statement number can be the entry point of a subprogram and a subprogram may have more than one entry point.

### (2) Body

The portion starting from the entry point and ending at the return point is called the body of the subprogram.

### (3) Return Point

The return point is provided to end the operation of the subprogram and return the control to the main program.

#### (4) Parameter

In the BASIC language, the notion of parameters is the most ambiguous. In fact, even the entry point of the subprogram can not be clearly defined. In practice, the main program and the subprograms are intermingled. In addition, as the range for variables is global in nature, any variable that can be changed during the subprogram execution may be considered as a parameter of the subprogram.

#### (5) The method of entry into a subprogram

GOSUB nnnn. Where nnnn is the entry point. For a detailed description please see 10.2.

## 10.2 GOSUB

In the usage of a subprogram, we will describe both the entry into the subprogram and the return to the main program.

### 10.2.1 GOSUB

GOSUB - Entry into a subprogram

Format:

GOSUB snum

Description:

The GOSUB statement is used to specify the entry point by the statement number snum. Upon execution, the system will record the statement number of the next statement following GOSUB on a stack, and the control of program execution is transferred to statement number snum.

Note:

If the statement number snum is not existent in the current program, the following error message will be displayed:

UL ERROR IN LINE

Execution mode:

Deferred mode

Remark:

The statements GOSUB and GOTO are quite similar in effect. The execution of GOSUB transfers the control of program execution to the subprogram, and this transfer is recorded on the Return-Address stack. When the program execution comes to the point of the RETURN statement, the control of the program execution will be transferred as indicated by the top of the Return-Address stack.

Upon the execution of each GOSUB statement, the return address will first be pushed onto the top of the Return-Address stack. And later upon the execution of a RETURN statement, a return address will be popped off the top of the stack, decrementing the height of Return-Address stack by 1 and execution goes back to the return address. If a RETURN statement is executed when the height of the Return-Address stack is zero, the following error message will be displayed:

RT ERROR IN LINE

For more detail on the usage of the stack, please see 10.3.

## 20.2.2 ON/GOSUB

ON/GOSUB - Computed entry into a subprogram

Format:

ON aexpr GOSUB snum-1{[,snum-i]}

Description:

The relation between ON/GOSUB and GOSUB statements is similar to that between ON/GOTO and GOTO statements.

When the value of the numeric expression `aexpr` is 1, the statement is equivalent to `GOSUB snum-1`. In general, when the value of `aexpr` is `i`, the statement is equivalent to `GOSUB snum-i`.

Note:

Like the `GOSUB` statement, if `snum-i` is not to be found anywhere in the current program, the following error message will be displayed:

UL ERROR IN LINE

Similarly, when the value of `aexpr` exceeds the number of `snum`, the following error message will be displayed:

SN ERROR IN LINE

Execution mode:

Deferred mode

Remark:

During the program execution, the error resulted from the absence of `snum-i` in the current program will be detected only when `aexpr` is equal to `i`.

### 10.2.3 RETURN

RETURN - Return point of the subprogram

Format:

RETURN

Description:

Upon the execution of the `RETURN` statement, the program execution will go back to the statement next to the last executed `GOSUB` statement.



Note:

When the number of execution of RETURN exceeds that of GOSUB, the following error message will be displayed:

RT ERROR IN LINE

Execution mode:

Deferred mode

Remark:

Please refer to 10.2.1 on GOSUB for a detailed description.

## 10.2.4 Examples

```
@NEW
@10 ?"SIZE :";
@20 INPUT N
@30 ?"USING KEY";
@40 INPUT A$
@50 C=ASCII(A$)
@60 FOR I = 0 TO N
@70 GOSUB 500
@80 NEXT I
@90 FOR I = N-1 TO 0 STEP-1
@100 GOSUB 500
@110 NEXT I
@120 END
@500 ?SPA(N-I);
@510 ?STR(I*2,C)
@520 RETURN
SIZE :?2
USING KEY?Y
```

```
YY
YYYY
YY
```

```
READY
@RUN
SIZE:?7
USING KEY?:
```

```

READY
@
@NEW
@10 INPUT N:N0=N
@20 GOSUB 100
@30 GOTO 10
@100 N=ABS(N)
@110 IF N0<0 THEN ?"-";
@120 GOSUB 200
@130 IF N>0 THEN 120
@140 PRINT N$:N$=""
@150 RETURN
@200 N1=INT(N/10)
@210 C=N-N1*10
@210 N$=CHR$(C+48)+N$
@230 N=N1
@240 RETURN
@RUN
? 1234
1234
? -78
-78
? -67
-67
778.78
78
?-667666.34
-667666
?67.9
67
? +678.6
678

```

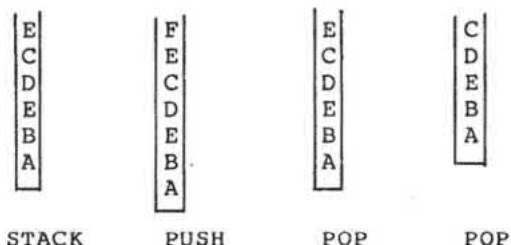
```
? -777.0
=777
?↑C
STOP AT LINE 10
```

```
READY
@
```

## 10.3 Recursive Subprogram

In 10.2.1 on GOSUB, we have described the usage of stack. We will go into the details in this respect in this section and then go on with the Recursive Subprogram comprised of stack and subprogram return point.

The stack is an ordered list, its insertion and deletion occur only at an opening which is called the TOP. The configure of stack is shown on the following page. The insertion and deletion at the top are called PUSH and POP respectively. A PUSH stores a new data onto the top of the stack so that the next data accessible becomes the new data and not the original top. A POP works in an opposite manner, it takes the data on the top off the stack, and the original next data becomes the new top of the stack.



To visualize the notion of a stack, think of the dishes placed in a pile as seen in a cafeteria. The customers take the dish off the top in sequence, and the washed dishes are placed onto the top of the pile. This phenomenon is a living analogy to PUSH and POP.

The operations of PUSH and POP on the stack function in a special format which is called first-in-last-out or last-in-first-out. In other words, all data is taken off the stack in an order contrary to that in which they are inserted. See the following configure:

```
PUSH A,B,C,D,E,F
```

```
STACK ABCDEF
```

TOP

```
POP F,E,D,C,B,A,
```

In fact, there is a Return-Address stack in the BASIC system. Upon the execution of a CALL statement, the system will PUSH the statement number next to the CALL statement onto the Return-Address Stack. Later in the program upon the execution of a RETURN statement, the system will POP the top of the Return-Address Stock and the control of the program will be transferred to the statement indicated by the top.

Try the following example:

```
@10 I=1
@20 I=I+1
@30 IF I<5 THEN GOSUB 20:"RET"
@40 PRINT I
@50 RETURN
@RUN
5
RET      5
RET      5

RT ERROR IN LINE 50
READY
```

In mathematics, many a function is expressed as a recursive function. Take  $n!$  as an example:

$$n! = \begin{cases} 1 & \text{if } n=0 \\ n*(n-1)! & \text{if } n>0 \end{cases}$$

To express this function in a recursive subprogram, obviously we need a data stack N in addition to the Return-Address Stack. In fact, the programmer must set up this N stack and define the operations of PUSH and POP. In the case of languages which do not have an automatic stack-manipulating feature, we usually resort to a one-dimensional array together with its index to simulate the function of a stack.

Examine the following definition:

Array : A(N)      Index : I

PUSH(X) : I=I+1 : A(I)=X

POP(X) : I=I-1

Listed below is a BASIC program with a recursive subprogram.

```

@10 INP"N=",N
@20 DIM S(N)
@30 I=0
@40 GOSUB 100
@50 PRINT P
@60 END
@100 I=I+1
@110 S(I)=N
@120 IF N=0 THEN P=1: RET
@130 N=N-1:GOSUB 100
@140 I=I-1:N=S(I)
@150 P=N*P
@160 RETURN
@RUN
N=?7
5040

READY
@RUN
N=?8
40320

READY
@

```

---

## Chapter 11

### User-Defined Function

---

In MPF-IP BASIC, there are a number of built-in functions in connection with numeric and string operations. In fact, the user can define functions themselves to facilitate programming. These user-defined functions have the same effect as the built-in functions in practical applications.

The name of a user-defined function begins with "FN" followed by a variable name which can either be a numeric variable name or be a string variable name. The resulting value of a user-defined function is a numeric value if the variable name is a numeric variable. Similarly is the case with the string variables.

A user-defined function is usually used as an operand of an expression in practical usage. An argument is usually added to the function name. This is in general an actual argument enclosed by parentheses. An actual argument may be a constant, a variable, a built-in function, or an expression. Sometime it can be another user-defined function.

In Appendix D, all the built-in functions are listed for reference.

## 11.1 DEF Statement

DEF - Define a function

Format:

```
DEF FN var-1(var-2)=expr
```

Description:

The DEF statement is used to define a user-defined function.

Note:

Only one dummy argument (var-2) is allowed in a user-defined function, it can either be a numeric variable or be a string variable. In addition, this dummy variable is merely a local variable which will not affect the value of another variable with the same variable name during program execution.

In a user-defined function, the type of var-1 must match the type of the expr On the righthand side of the equal sign (=). The var-1 must be a generally accepted legal variable name.

Description:

From Format, we can derive the following two fundamental types:

(1) DEF FN avar (var)=aexpr

(2) DEF FN svar (var)=sexpr

In (1) a numeric function is defined while in (2) a string function is defined.

In the above, expr represents an expression, the operands in which can be a constant, variable, built-in function or another user-defined function. Be sure, however, not to insert the function itself in expr. The execution of a function defined in this way will bring the system into an infinite loop which will never come to an end. In this case, the only way to recovery is to press the RESET key to reinitialize the system.

Try the following example:

```
@NEW
@10 DEF FNA(F)=F+8
@20 DEF FNB(U)=U+100
@30 DEF FNC(U)=U+FNA(U)
@40 INPUT A
@50 PRINT FNA(A);FNB(A);FNC(A)
@60 GOTO 40
@RUN
?12
  20  112  124
?30
  38  130  160
?699
  707  799  1498
?↑C
STOP AT LINE 30
```

READY

Like REM, DATA, the DEF statement is a non-executable statement. It is also one of the declaration statements like DIM (For details please see chapter 7). It doesn't matter where in the program it is located.

Try the following example:

```
@NEW
@10 FOR I=1 TO 5
@20 PRINT FNS1(PI/I)
@30 NEXT I
@40 END
@100 DEF FNS1(J)=1/SIN(J)
@RUN
8.38861E+06
1
1.1547
1.41421
1.7013
```

READY

@



One must take note not to have more than two variable names following FN to be identical. The system will not tell the user that it is illegal but among the user-defined functions with the same name, only that with the lowest statement number will be acknowledged by the system.

Try the following example:

```
@NEW
@10 DEF FNA$(A$)=RIGHT$(A$,4)
@20 DEF FNA$(A$)=LEFT$(A$,4)
@30 INPUT"STRING",A$
@40 PRINT A$;FNA$(A$)
@50 GOTO 30
@RUN
STRING?ABCDEFGH
ABCDEFGHDEFGH
STRING?BASIC-MPF-IP
BASIC-MPF-IPIC-MPF-IP
STRING?
```

## 11.2 Usage of User-Defined Function

The user-defined function greatly facilitates programming. When the function is repeatedly used in a program, the time for inputting the program is greatly decreased and the number of errors resulted therefrom is also diminished. In addition, the memory space of the system is saved to a considerable extent. And most importantly, it is the technique provided for modular programming.

Listed below are some trigonometric functions which are not provided by the BASIC-MPF-IP system but can be derived from other functions.

(1) Cotangent: COT (X)

```
DEF FNA(X)=1/TAN(X)
```

(2) Secant : SEC(X)

```
DEF FNA(X)=1/COS(X)
```

- (3) Cosecant : CSC(X)  
 $\text{DEF FNA}(X) = 1/\text{SIN}(X)$
- (4) Arccotangent:  $\text{COT}^{-1}(X)$   
 $\text{DEF FNA}(X) = \text{PI}/2 - \text{ATN}(X)$
- (5) Hyperbolic Sine: SINH(X)  
 $\text{DEF FNA}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
- (6) Hyperbolic Cosine: COSH(X)  
 $\text{DEF FNA}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
- (7) Hyperbolic Tangent: TANH(X)  
 $\text{DEF FNA}(X) = (\text{EXP}(X) - \text{EXP}(-X)) / (\text{EXP}(X) + \text{EXP}(-X))$
- (8) Hyperbolic Cotangent : COTH(X)  
 $\text{DEF FNA}(X) = (\text{EXP}(X) + \text{EXP}(-X)) / (\text{EXP}(X) - \text{EXP}(-X))$
- (9) Hyperbolic Secant: SECH(X)  
 $\text{DEF FNA}(X) = 2 / (\text{EXP}(X) + \text{EXP}(-X))$
- (10) Hyperbolic Cosecant: CSCH(X)  
 $\text{DEF FNA}(X) = 2 / (\text{EXP}(X) - \text{EXP}(-X))$
- (11) Archhyperbolic Sine:  $\text{SINH}^{-1}(X)$   
 $\text{DEF FNA}(X) = \text{LN}(X + \text{SQU}(X*X+1))$
- (12) Archhyperbolic Cosine:  $\text{COSH}^{-1}(X)$   
 $\text{DEF FNA}(X) = \text{LN}(X + \text{SQR}(X*X-1))$
- (13) Archhyperbolic Tangent:  $\text{TANH}^{-1}(X)$   
 $\text{DEF FNA}(X) = \text{LN}((1+X)/(1-X))/2$
- (14) Archhyperbolic Cotangent:  $\text{COTH}^{-1}(X)$   
 $\text{DEF FNA}(X) = \text{LN}((X+1)/(X-1))/2$

(15) Archyperbolic Secant: SECH-1(X)

DEF FNA(X)=LN(1/X+SQR(1/(X\*X)-1))  
-1

(16) Archyperbolic Cosecant: CSCH (X)

DEF FNA(X)=LN(1/X+SQR(1/(X\*X)+1))

---

## Chapter 12

### Combination with Non-BASIC Program

---

Undoubtedly MPF-IP BASIC can execute a pure BASIC program. In addition, it can execute a portion of assembly program.

In the execution of the assembly program, the absolute-address variable is also used. In connection with I/O, we have described the INP function and OUT statement in chapter 9.

In connection with the execution of an assembly program. We can use the CALL statement. For the usage of absolute address, we have the PEEK function and the POKE statement.

#### 12.1 CALL Statement

MPF-IP uses Z-80A which is one of the most prevalent in the market as its CPU(Central Processing Unit).

As a result, among the various assembly languages, Z-80 Assembly is the only one executable on the MPF-IP BASIC system.

CALL - Execution of an assembly program

Format:

CALL aexpr

### Description:

Upon the execution of the CALL statement, firstly the value of aexpr is evaluated and then the control of the program is transferred to the absolute address aexpr for the execution of a portion of assembly program.

### Execution mode:

Immediate and Deferred mode

### Remark:

The execution of assembly program will go on until a RET (mnemonic:C9 Hex=201) is encountered and then the execution will go back to the BASIC program. There is only one RET in the most simple assembly program.

### Try the following examples:

```
@10 POKE 61660,201
@20 FOR I=1TO5
@30 CALL61660
@40 PRINT I
@50 NEXT I
@RUN
1
2
3
4
5
```

```
READY
@NEW
A=15020
@B=A
@10 A=B:B=B+6
@20 CALL A
@30 PRINT "HAHA"
@XEQ
RG ERROR IN LINE 20
```

```
READY
@XEQ
UL ERROR IN LINE 20
```

READY  
@XQR  
OF ERROR IN LINE 20

READY  
@XEQ  
ST ERROR IN LINE 20

READY  
@  
SN ERROR IN LINE 20

READY  
@XEQ  
RT ERROR IN LINE 20

READY  
@XEQ  
DA ERROR IN LINE 20

READY  
@XEQ  
NX ERROR IN LINE 20

READY  
@XEQ  
CV ERROR IN LINE 20  
READY  
@XEQ  
CK ERROR IN LINE 20

READY  
@XEQ  
FN ERROR IN LINE 20

READY  
@XEQ  
DW ERROR IN LINE 20

READY  
@XEQ  
DS ERROR IN LINE 20

READY  
@XEQ  
OV ERROR IN LINE 20  
HAHA

```
READY
@XEQ
HAHA
```

```
READY
@XEQ
  ERROR IN LINE 20
*****MPF-I-PLUS*****
<
```

**Note:**

The value of aexpr must be in the range zero to 65535, otherwise the following error message will be displayed:

```
OV ERROR IN LINE
```

Try the following example:

```
@CALL -1
OV ERROR IN LINE
FE00 AF B1B0 BC 9800
<
```

## 12.2 POKE/PEEK

The POKE statement and the PEEK function is used in connection with the absolute address variable. As an absolute variable, a number is restricted to be in the range zero to 255, occupying a byte in the memory.

**POKE** - Assign a value to an absolute address

**Format:**

```
POKE aexpr-1, aexpr-2
```

**Description:**

The POKE statement is used to assign the value of aexpr-2 to the absolute address aexpr-1.

**Execution mode:**

Immediate and deferred mode

Note:

The value of `aexpr-1` must be between zero and 65535, otherwise the following error message will be displayed:

OV ERROR IN LINE

As Z-80A is used as the CPU on the MPF-IP system, the absolute address ranges from zero through 65535. However, the actual memory on MPF-IP may be less, thus the usage of some address may become insignificant. In addition, to the Read Only Memory of MPF-IP, the POKE statement has no effect. The value of `aexpr-2` is expected to fall between zero and 255. If `aexpr-2` exceeds 255 the actual value in effect will be `INT(aexpr-2/256)`. If `INT(aexpr-2/256)` is a gain greater than 255, the INT operation is repeated.

Try the following examples:

```
@NEW
@10 I=256
@20 POKE 61660,I
@30 PRINT INT(I/256),I,PEEK(61660)
@40 I=I+256
@50 GOTO 20
@RUN
1      256      1
2      512      2
3      768      3
4     1024      4
5     1280      5
6     1536      6
7     1792      7
8     2048      8
.
.
.
.
@NEW
@10 I=143
@20 POKE 61660,I
@30 PRINT INT(I/256),I,PEEK(61660)
@40 I=I+256
@50 GOTO 20
```



|      |      |    |
|------|------|----|
| @RUN |      |    |
| 1    | 399  | 1  |
| 2    | 655  | 2  |
| 3    | 911  | 3  |
| 4    | 1167 | 4  |
| 5    | 1423 | 5  |
| 6    | 1679 | 6  |
| 7    | 1935 | 7  |
| 8    | 2191 | 8  |
| 9    | 2447 | 9  |
| 10   | 2703 | 10 |
| 11   | 2959 | 11 |
| .    |      |    |
| .    |      |    |
| .    |      |    |
| .    |      |    |

PEEK - Read a numeric value from an absolute address

Format:

PEEK (aexpr)

Description:

The PEEK function returns the value currently stored in the absolute address specified by aexpr. The value will be between zero and 65535, otherwise the following error message will be displayed:

OV ERROR IN LINE

Try the following example:

```
@?PEEK(-1)
OV ERROR IN LINE
129
@
```

And now have fun trying the following example:

```
@RESET
*****MPF-I-PLUS*****
CONTROL B
BASIC-IP, ORG:F000
@10 I=15*16*16*16
@20 J=PEEK(I)
@30 IF J< 32 THEN 70
@40 IF J> 96 THEN 70
@50 PRINT CHR$(J);
@60 IF POS(0)>19 THEN PRINT
@70 I=I+1
@80 GOTO 20
@RUN
I=15*16*16*16 J=PEEK(I)0
IF J< 32 THEN 70@ IF J>96
THEN 70 P PRINT CHR$(J);A
IF POS(0)>19 THEN
PRINT I=I+1 GOTO 20??
>EW
CONTROL C
READY
@
```

---

## Appendix A    ASCII Characters

---

| <u>Decimal<br/>Value</u> | <u>Hexadecimal<br/>Value</u> | <u>Meaning<br/>(Abbreviation)</u> |
|--------------------------|------------------------------|-----------------------------------|
| 0                        | 00                           | Null<br>(NUL)                     |
| 1                        | 01                           | Start of<br>Heading<br>(SOH)      |
| 2                        | 02                           | Start of<br>Text<br>(STX)         |
| 3                        | 03                           | End of<br>Text<br>(ETX)           |
| 4                        | 04                           | End of<br>Transmission<br>(ET)    |
| 5                        | 05                           | Enquiry<br>(ENQ)                  |
| 6                        | 06                           | Acknowledge<br>(ACK)              |
| 7                        | 07                           | Bell<br>(BEL)                     |
| 8                        | 08                           | Backspace<br>(BS)                 |
| 9                        | 09                           | Horizontal<br>Tabulation<br>(HT)  |
| 10                       | 0A                           | Line Feed<br>(LF)                 |
| 11                       | 0B                           | Vertical<br>Tabulation<br>(VT)    |

|    |    |                                          |
|----|----|------------------------------------------|
| 12 | 0C | Form Feed<br>(FF)                        |
| 13 | 0D | Carriage<br>Return<br>(CR)               |
| 14 | 0E | Shift Out<br>(SO)                        |
| 15 | 0F | Shift In<br>(SI)                         |
| 16 | 10 | Data Link<br>Escape<br>(DLE)             |
| 17 | 11 | Device<br>Control 1<br>(DC1)             |
| 18 | 12 | Device<br>Control 2<br>(DC2)             |
| 19 | 13 | Device<br>Control 3<br>(DC3)             |
| 20 | 14 | Device<br>Control 4<br>(DC4)             |
| 21 | 15 | Negative<br>Acknowledge<br>(NAK)         |
| 22 | 16 | Synchronous<br>Idle<br>(SYN)             |
| 23 | 17 | End of<br>Transmission<br>Block<br>(ETB) |

|    |    |                             |
|----|----|-----------------------------|
| 24 | 18 | Cancel<br>(CAN)             |
| 25 | 19 | End of<br>Medium<br>(EM)    |
| 26 | 1A | Substitute<br>(SUB)         |
| 27 | 1B | Escape<br>(ESC)             |
| 28 | 1C | File Separator<br>(FS)      |
| 29 | 1D | Group<br>Separator<br>(GS)  |
| 30 | 1E | Record<br>Separator<br>(RS) |
| 31 | 1F | Unit<br>Separator<br>(US)   |
| 32 | 20 | Space                       |
| 33 | 21 | Exclamation<br>Point<br>(!) |
| 34 | 22 | Quotation<br>Mark<br>(")    |
| 35 | 23 | Number<br>Sign<br>(#)       |
| 36 | 24 | Dollar<br>Sign<br>(\$)      |

|    |    |                                     |
|----|----|-------------------------------------|
| 37 | 25 | Percent<br>Sign<br>(%)              |
| 38 | 26 | Ampersand<br>(&)                    |
| 39 | 27 | Apostrophe<br>(')                   |
| 40 | 28 | Opening<br>Parenthesis<br>( $<$ )   |
| 41 | 29 | Closing<br>Parenthesis<br>( $>$ )   |
| 42 | 2A | Asterisk<br>(*)                     |
| 43 | 2B | Plus<br>(+)                         |
| 44 | 2C | Comma<br>(,)                        |
| 45 | 2D | Hyphen<br>(Minus)<br>(-)            |
| 46 | 2E | Period<br>(Decimal<br>Point)<br>(.) |
| 47 | 2F | Slant<br>(/)                        |
| 48 | 30 | Zero<br>(0)                         |
| 49 | 31 | One<br>(1)                          |

|    |    |                         |
|----|----|-------------------------|
| 50 | 32 | Two<br>(2)              |
| 51 | 33 | Three<br>(3)            |
| 52 | 34 | Four<br>(4)             |
| 53 | 35 | Five<br>(5)             |
| 54 | 36 | Six<br>(6)              |
| 55 | 37 | Seven<br>(7)            |
| 56 | 38 | Eight<br>(8)            |
| 57 | 39 | Nine<br>(9)             |
| 58 | 3A | Colon<br>(:)            |
| 59 | 3B | Semicolon<br>(;)        |
| 60 | 3C | Less Than<br>(<)        |
| 61 | 3D | Equals<br>(=)           |
| 62 | 3E | Greater<br>Than<br>(>)  |
| 63 | 3F | Question<br>Mark<br>(?) |
| 64 | 40 | Commercial<br>At<br>(@) |

|    |    |                    |
|----|----|--------------------|
| 65 | 41 | Uppercase A<br>(A) |
| 66 | 42 | Uppercase B<br>(B) |
| 67 | 43 | Uppercase C<br>(C) |
| 68 | 44 | Uppercase D<br>(D) |
| 69 | 45 | Uppercase E<br>(E) |
| 70 | 46 | Uppercase F<br>(F) |
| 71 | 47 | Uppercase G<br>(G) |
| 72 | 48 | Uppercase H<br>(H) |
| 73 | 49 | Uppercase I<br>(I) |
| 74 | 4A | Uppercase J<br>(J) |
| 75 | 4B | Uppercase K<br>(K) |
| 76 | 4C | Uppercase L<br>(L) |
| 77 | 4D | Uppercase M<br>(M) |
| 78 | 4E | Uppercase N<br>(N) |
| 79 | 4F | Uppercase O<br>(O) |
| 80 | 50 | Uppercase P<br>(P) |



|    |    |                           |
|----|----|---------------------------|
| 81 | 51 | Uppercase Q<br>(Q)        |
| 82 | 52 | Uppercase R<br>(R)        |
| 83 | 53 | Uppercase S<br>(S)        |
| 84 | 54 | Uppercase T<br>(T)        |
| 85 | 55 | Uppercase U<br>(U)        |
| 86 | 56 | Uppercase V<br>(V)        |
| 87 | 57 | Uppercase W<br>(W)        |
| 88 | 58 | Uppercase X<br>(X)        |
| 89 | 59 | Uppercase Y<br>(Y)        |
| 90 | 5A | Uppercase Z<br>(Z)        |
| 91 | 5B | Opening<br>Bracket<br>([) |
| 92 | 5C | Reverse<br>Slant<br>(\)   |
| 93 | 5D | Closing<br>Bracket<br>(]) |
| 94 | 5E | Circumflex<br>(^)         |
| 95 | 5F | Underscore<br>(-)         |

|     |    |                        |
|-----|----|------------------------|
| 96  | 60 | Grave<br>Accent<br>(`) |
| 97  | 61 | Lowercase a<br>(a)     |
| 98  | 62 | Lowercase b<br>(b)     |
| 99  | 63 | Lowercase c<br>(c)     |
| 100 | 64 | Lowercase d<br>(d)     |
| 101 | 65 | Lowercase e<br>(e)     |
| 102 | 66 | Lowercase f<br>(f)     |
| 103 | 67 | Lowercase g<br>(g)     |
| 104 | 68 | Lowercase h<br>(h)     |
| 105 | 69 | Lowercase i<br>(i)     |
| 106 | 6A | Lowercase j<br>(j)     |
| 107 | 6B | Lowercase k<br>(k)     |
| 108 | 6C | Lowercase l<br>(l)     |
| 109 | 6D | Lowercase m<br>(m)     |
| 110 | 6E | Lowercase n<br>(n)     |
| 111 | 6F | Lowercase o<br>(o)     |

|     |    |                                 |
|-----|----|---------------------------------|
| 112 | 70 | Lowercase p<br>(p)              |
| 113 | 71 | Lowercase q<br>(q)              |
| 114 | 72 | Lowercase r<br>(r)              |
| 115 | 73 | Lowercase s<br>(s)              |
| 116 | 74 | Lowercase t<br>(t)              |
| 117 | 75 | Lowercase u<br>(u)              |
| 118 | 76 | Lowercase v<br>(v)              |
| 119 | 77 | Lowercase w<br>(w)              |
| 120 | 78 | Lowercase x<br>(x)              |
| 121 | 79 | Lowercase y<br>(y)              |
| 122 | 7A | Lowercase z<br>(z)              |
| 123 | 7B | Opening (left)<br>Brace<br>({)  |
| 124 | 7C | Vertical<br>Line<br>( )         |
| 125 | 7D | Closing (right)<br>Brace<br>(}) |
| 126 | 7E | Tilde<br>(~)                    |
| 127 | 7F | Delete<br>(DEL)                 |

---

## Appendix B

### MPF-IP BASIC Statements

---

| Name      | Abbreviaton | Description                           | Reference |
|-----------|-------------|---------------------------------------|-----------|
| CALL      | CAL         | Execution of an assembly program      | 12.1      |
| DATA      | DAT         | Stock of data                         | 9.3       |
| DEF       | DEF         | User-defined functions                | 11.1      |
| DIM       | DIM         | Declaration of an array               | 7.1       |
| END       | END         | End the program execution             | 4.2       |
| FOR       | FOR         | FOR-loops                             | 5.1.1     |
| GOSUB     | GOS         | Jump to a subprogram                  | 10.1.1    |
| GOTO      | GOT         | Unconditional control transfer        | 5.3.1     |
| IF..THEN  | IF..THE     | Conditional Control Transfer          | 5.2.1     |
| INPUT     | INP         | Input through the keyboard            | 9.2       |
| LET       |             | Assignment statement                  | 4.1       |
| NEXT      | NEX         | Next FOR-loop                         | 5.1.2     |
| ON..GOSUB | ON..GOS     | Computed jump to a subprogram         | 10.1.2    |
| ON..GOTO  | ON..GOT     | Computed control transfer             | 5.4.1     |
| OUT       | OUT         | Send to output port                   | 9.5       |
| POKE      | POK         | Assign a value to an absolute address | 12.2      |
| PRINT     | ?(PRI)      | Output to the display                 | 9.1       |

|           |     |                                   |        |
|-----------|-----|-----------------------------------|--------|
| RANDOMIZE | RAN | Reset the random number cycle     | 4.4    |
| READ      | REA | Read an entry from DATA statement | 9.3    |
| REM       | !   | Remark                            | 4.3    |
| RESTORE   | RES | Reset the DATA point              | 9.3    |
| RETURN    | RET | Return to the main program        | 10.1.3 |
| STOP      | STO | Stop the program execution        | 4.2    |

---

## Appendix C

### MPF-IP BASIC Commands

---

| Name     | Abbreviation | Description                                      | Reference |
|----------|--------------|--------------------------------------------------|-----------|
| CONTINUE | CON          | Continue the execution of an interrupted program | 3.1.2     |
| EDIT     | EDI          | Edit program statements                          | 3.2.3     |
| FREE     | FRE          | Show the memory available                        | 3.4.1     |
| GOTO     | GOT          | Change the start point of program execution      | 3.1.1     |
| HEX      | HEX          | Hexadecimal conversion                           | 3.4.2     |
| LIST     | LIS          | List a program                                   | 3.2.1     |
| LOAD     | LOA          | Read a program from a magnetic tape              | 3.3.2     |
| NEW      | NEW          | Clear a program                                  | 3.2.2     |
| NEW      | NEW*         | Clear a program                                  | 3.2.2     |
| QUIT     | QUI          | Return to the monitor                            | 3.1.3     |
| RUN      | RUN          | Execute a program                                | 3.1.1     |
| SAVE     | SAV          | Save a program on the magnetic tape              | 3.3.1     |
| XEQ      | XEQ          | Execute a program                                | 3.1.1     |

---

## Appendix D

### MPF-IP BASIC Built-in Functions

---

The followed is a list of MPF-IP BASIC built-in functions in alphabetical order. Please note that the argument of all the trigonometric functions is expressed in radians (1 radian= $180/\pi$  degrees). In this list, X represents a numeric expression while S represents a string expression.

| Name(Argument) | Abbreviation | Description                                                | Reference |
|----------------|--------------|------------------------------------------------------------|-----------|
| ABS(X)         | ABS(X)       | Absolute value                                             | 6.2.1     |
| ASCII(S)       | ASC(X)       | ASCII code of the first character of the string expression | 8.4.1     |
| ATN(X)         | ATN(X)       | Arctangent function<br>Results is in radians               | 6.2.12    |
| CHR\$(X)       | CHR(X)       | Corresponding character according to the ASCII code        | 8.4.2     |
| COS(X)         | COS(X)       | Cosine function                                            | 6.2.10    |
| EXP(X)         | EXP(X)       | Exponentiation with base $e=2.71828$                       | 6.2.2     |
| INSTR(X,S1,S2) | INS(X,S1,S2) | Find the position of presence of S2 in S1 starting from X. | 8.4.3     |
| INT(X)         | INT(X)       | The Gaussian (Integer) function                            | 6.2.3     |
| INP(X)         | INP(X)       | Input data from port X                                     | 9.4       |

|                 |              |                                                                           |        |
|-----------------|--------------|---------------------------------------------------------------------------|--------|
| LEFT\$(S,X)     | LEF(S,X)     | The leftmost X characters of S.                                           | 8.4.4  |
| LEN(S)          | LEN(S)       | Length of string S.                                                       | 6.2.4  |
| LN(X)           | LN(X)        | Natural logarithm                                                         | 6.2.5  |
| MID\$(S,X1,X2)  | MID(S,X1,X2) | The X2 characters starting from (X1)th character of string S              | 8.4.6  |
| NUM\$(X)        | NUM(X)       | Convert X to a string                                                     | 8.4.7  |
| PEEK(X)         | PEE(X)       | Read data from absolute address X                                         | 12.2   |
| RIGHT\$(S,X)    | RIG(S,X)     | The rightmost characters starting from (X)th character of string          | 8.4.8  |
| RND(X)          | RND(X)       | Random number function                                                    | 6.2.6  |
| SGN(X)          | SGN(X)       | Sign function                                                             | 6.2.7  |
| SIN(X)          | SIN(X)       | Sine function                                                             | 6.2.9  |
| SPACE\$(X)      | SPA(X)       | String composed of spaces                                                 | 8.4.9  |
| SQR(X)          | SQR(X)       | Square root function                                                      | 6.2.8  |
| STRING\$(X1,X2) | STR(X1,X2)   | String composed of X1 characters with the corresponding ASCII code of X2. | 8.4.10 |
| TAN(X)          | TAN(X)       | Tangent function                                                          | 6.2.11 |
| VAL(X)          | VAL(X)       | Value of the numeric string S                                             | 8.4.11 |



---

## Appendix E

### MPF-IP BASIC Error Messages

---

The format of MPF-IP BASIC error messages is as follows:

|    |                                                                                                                                                                                                        |
|----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| XX | ERROR IN LINE                                                                                                                                                                                          |
| XX | Description                                                                                                                                                                                            |
| UL | A statement number not found in the program is referred to.                                                                                                                                            |
| OF | (1) LOG(X) X<=0<br>(2) SQR(X) X>=0<br>(3) Division by zero                                                                                                                                             |
| RT | A RETURN statement without the corresponding GOSUB                                                                                                                                                     |
| DA | (1) Items in DATA statements are insufficient.<br>(2) Upon the execution of EDIT, the string to be modified is not found in the program.                                                               |
| NX | A NEXT statement without the corresponding FOR                                                                                                                                                         |
| CV | Conversion errors                                                                                                                                                                                      |
| UN | (1) Underflow<br>(2) Errors in evaluations of numeric values.                                                                                                                                          |
| SN | (1) Syntax error<br>(2) The maximum memory is extended when the array variables are redeclared<br>(3) The values of subscripts run out of range<br>(4) Absence of parameters in the usage of functions |
| ST | (1) Incorrect usage of parameters in functions<br>(2) Too many folds of parentheses are used in an expression                                                                                          |

- OV
- (1) Overflow
  - (2) Errors in evaluation of numeric values
  - (3) Errors in the usage of absolute address
  - (4) The value of aexpr is negative in the usage of the LEFT\$ function
  - (5) The value of aexpr is negative in the usage of The TAB function
- RG
- (1) The values of subscripts exceed the range
  - (2) Run out of memory

---

## Appendix F

### Fundamental Definitions

---

#### \* Syntactic Definitions and Abbreviations

The following definitions are based upon Backus-Naur Form (BNF). These usages are not available in BASIC. In BNF, some special symbols such as {and| can clearly express the structure or relation. The symbol := is used to mark the beginning of the definition on the lefthand side. These special symbols are called Metasymbols.

| Used to separate the alternatives

[] Used to indicate the item is optional

{ } Used to indicate the item can be repeated

~ Used to indicate a necessary blank

metasymbol

:= |[ ]|[ { } ]|~

digit

:= 1|2|3|4|5|6|7|8|9|0

letter

:= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

special

:= !|~|~|\$|%|&|.|(|)|~|\*|:=|~|-|@|+|;|?|/|>|]|<|[[|^|~|"

character

:= letter|digit|special

alphanumeric

:= letter|digit

name

:= letter[*digit*]

numeric variable name

:= name

number

:= [*+*|*-*][*digit*]\*.[*digit*]\*[*E*[*+*|*-*]*digit*[*digit*]]

arithmetic variable

:= avar

avar

:= name

delimiter

:= |(*)*|=*-*|*+*|*^*|*<*|*>*|/*\**|,*;*|:

arithmetic operator

:= aop

aop

:= *+*|*-*|*\**|/*^*

arithmetic expression

:= aexpr

aexpr

:= avar|number

:= (aexpr)

:= [*+*|*-*] aexpr

:= avar subscript

Subscript

:= (aexpr {, aexpr})

```

avar
    := avar subscript
string
    := "[{character}]"
null string
    := ""
string variable
    := svar

svar
    := name$
string operator
    := sop

sop
    := +

string expression
    := sexpr

sexpr
    := svar|string
    := sexpr sop sexpr

string relational operator
    := srop

srop
    := =|>|>=|<=|<|<>

```

variable

:= var

var

:= avar | svar

expression

:= expr

expr

:= aexpr | sexpr

statement number

:= snum

snum

:= {digit}

line

:= snum[{instructions}] instruction

arithmetic relational operator

:= arop

arop

:= = | > | < | >= | <= | <>

arithmetic relational expression

:= arexpr

arexpr

:= aexpr arop aexpr

string relational expression

:= srexpr

```
srepr
    := sexpr srop sexpr
relational expression
    := rexpr
rexpr
    := srepr
    := arexpr
```

---

## Appendix I

### Some Subprograms in the Monitor

---

#### \*I-1

| Entry point(hex) | Decimal | Function    |
|------------------|---------|-------------|
| 0195             | 405     | PRT CONTROL |

#### Description:

This subprogram serves as a soft switch for the printer in BASIC. It has the same effect as CONTROL P in the monitor program execution.

Try the following example:

```
@NEW
@CALL 405
      PRT OFF@^
@10 CALL 405
@20 FOR I=1 TO 4
@30 PRINT SQR(I)
@40 NEXT I
@50 CALL 405
```

#### \*I-2

| Entry point(hex) | Decimal | Function     |
|------------------|---------|--------------|
| 01A9             | 425     | BEEP CONTROL |

#### Description:

This subprogram serves as a soft switch for the speaker in BASIC. It has the same effect as CONTROL G in the monitor program execution.

Try the following example:

```
@CALL 425
@CALL 425
```



Before the execution of the first CALL 425 subprogram, a beep is generated each time a character is entered through the keyboard. After the execution of the first CALL 425 subprogram, this beep-generating function is suspended. The function is recovered with the execution of the second CALL 425 subprogram.

\*I-3

| Entry point (hex) | Decimal | Function |
|-------------------|---------|----------|
| 07F6              | 2038    | CLRBUF   |

Description:

Upon the execution of this subprogram, all characters on the indicator panel are cleared and the prompt < is displayed.

Try the following example:

```

@NEW
@HEX FE27
  65063
@10 IF POS(0)>=015 THEN GOSUB 100
@20 PRINT I;
@30 I=I+1
@40 GOTO 10
@100 POKE 65063,0
@110 CALL 2038
@120 RETURN
@RUN
  0  1  2  3  4
< 5  6  7  8  9
< 10 11 12 13
< 14 15 16 17
< 18 19 20 21
< 22 23 24 25
< 26 ↑C
STOP AT LINE 30

READY
@

```

During the execution, the output linefeed is directly controlled by the program, hence the indicator panel keeps flashing with the output going on.

In the example, statement 100 POKE 65063,0 is used to reset the value of the POS function.

---

## Appendix G

### Ways to Save Memory

---

Listed below are ways to save memory occupied by a program in MPF-IP BASIC system:

- (1) Use "!" instead of REM. Two bytes are saved.
- (2) Use "?" instead of PRINT. Four bytes are saved.
- (3) Use ":" whenever possible. This can decrease the use of statement numbers.
- (4) Use the abbreviated form of reserved words.

In MPF-IP BASIC, the first three characters are used as the identifiable abbreviated form of a reserved word. The set of reserved words in MPF-IP BASIC includes commands, statements and built-in functions as listed in Appendix B, Appendix C and Appendix D. Followed is a list of some examples.

|              |       |            |
|--------------|-------|------------|
| CONTINUE     | <---> | CON        |
| RESTORE      | <---> | RES        |
| GOTO         | <---> | GOT        |
| ASCII(X)     | <---> | ASC        |
| RIGHT(X\$,A) | <---> | RIG(X\$,A) |

---

## Appendix H

### Library Constant

---

In 6.2 on numeric functions, we have described some built-in functions provided in MPF-IP BASIC. Among them, some are trigonometric functions in which the constant  $\pi$  is often used. As a result, the value of  $\pi$  is set by the system.

Try the following examples:

```
@PRINT PI
3.14159
@PRINT SIN(PI/2)
.999997
@PRINT COS(PI/2)
1.19209E-07
```



## **Multitech** INDUSTRIAL CORP.

OFFICE/9FL, 266 SUNGCHING ROAD, TAIPEI 104 TAIWAN, R.O.C.  
TEL: (02)551-1101 TELEX: "19162 MULTIC" FAX: 02551-1102  
FACTORY/1 INDUSTRIAL E. RD. 111, HSINCHU SCIENCE BASE  
INDUSTRIAL PARK, HSINCHU, TAIWAN, R.O.C.