OKI

MP Macroprocessor
User's Manual

Program Development Support Software

FIRST EDITION
ISSUE DATE:Aug. 1993

NOTICE

Preface

This manual describes the operation and macroprocessor language of the MP macroprocessor.

MP operates on Microsoft MS-DOS version 2.11 or later. A development system with the following hardware is needed to run MP.

- ■ Memory
  At least 70 Kbytes of memory are needed to run MP.
- ■ One floppy disk drive
  Operation is easier with two drives.
- ■ Monitor
- ■ Printer

This manual uses several symbols to make explanations easier to understand. These symbols and their general meanings are shown below. Wherever the symbols are used with different meanings, the alternative meanings will be explicitly indicated.

[ ]	The contents within the brackets are items to be input as necessary.
. . .	This indicates items that are to be input repeatedly as necessary.

Contents

1. Introduction

1.1 Functional Overview

The MP macroprocessor is a preprocessor that analyzes macros coded in source text and executes text expansion where appropriate.

MP recognizes only macroprocessor language when performing text expansion. Text portions other than macroprocessor language are simply copied as source text.

The output files created by MP are called expanded files. These expanded files are normally used as assembler source in the following step. Figure 1-1 shows development flow using MP.

$$\begin{tikzcd}
\text{Source text} \arrow[r] & \boxed{\text{MP}} \arrow[r] & \text{Expanded text} \arrow[r] & \boxed{\text{Assembler}}
\end{tikzcd}$$

Figure 1-1. Development Flow Using MP

The MP macroprocessor provides a set of standard macros with numerous functions as tools for defining user macro instructions. For example, if a frequently used sequence of instructions in a program is defined as a macro, then only the macro needs to be coded, thereby reducing program code. The MP macroprocessor provides native macro commands called standard macros, which the user can use to define his own macros. In addition to simple symbol definition and text string control, standard macros provide file include functions and text expansion control, as described below.

With the INCLUDE standard macro, the contents of a different text file can be inserted at any place in the source text. With the MACROLIB standard macro, user-defined macros can be managed as libraries. By using the text expansion control macros, such as IF and IFDEF, macro expansion control and repeated text generation is possible within source text. Generation of source text based on a text module structure is easy with MP.

1.1.1 Macro Functions

The MP macroprocessor provides the macros shown in Table 1-1 as standard macros.

Table 1-1. MP Standard Macros

Category	Macro Name	Description
User macro definitions	MACRO	Defines user-defined macro symbols, parameters, and contents.
Symbol definitions	DEFINE SET MATCH	Assigns character strings and values of hexadecimal expressions to symbols.

(continued on next page)

Table 1-1(Cont). MP Standard Macros

Category	Macro Name	Description
String comparisons	EQS NES LTS LES GTS GES	Compares specified strings in text and returns the result as true (-1) or false (0).
String operations	EVAL LEN SUBSTR	Returns the hexadecimal string expansion of an expression's value, the length of text, or a partial string from some text.
Expansion control functions	IF IFDEF IFUNDEF WHILE REPEAT EXIT	Controls flow of macro expansion of text by evaluating expressions or symbols.
File include functions	INCLUDE MACROLIB	Inserts specified file contents into source text. If the inserted text contains INCLUDE and MACROLIB, then the further specified contents will be inserted. Nesting up to 13 levels is possible.
Listing functions	GEN GENONLY	Specifies the output format of text expanded by a call or macro expansion.
Other	METACHAR IN OUT COLOR SYSTEM EXIST SOURCE FDATE FTIME DATE TIME PURGE ALLPURGE	1. Controls exception processing of macro expansions of comments, brackets, and escapes. 2. Controls console I/O, operating system, file name, date, time, and symbol definition purges.

1.1.2 Macro Application Examples

The following simplified examples describe how macros are used.

First consider a program that transfers a continuous block of data from some address to a different data area. This program could be written in assembler source text as follows.

```
        MOV     X1,#0
        MOVB    R2,#COUNT
ACT:
        MOVB    A,FROM[X1]
        MOVB    TO[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT
```

The same program might be coded several times by changing the values of the symbols COUNT, FROM, and TO in the above listing. If one creates a user-defined macro using the standard macro MACRO, then coding of source text will be greatly simplified. An example of code of the above source text using a macro is shown below.

```
@MACRO(SEND(FROM,TO,COUNT))LOCAL ACT(
        MOV     X1,#0
        MOVB    R2,#@COUNT
@ACT:
        MOVB    A,@FROM[X1]
        MOVB    @TO[X1],A
        INC     X1
        DECB    R2
        JC      NE,@ACT
)
```

The use of the above macro within source text is known as calling the macro. To call a macro, code as shown below.

```
@SEND(100,200,3)
@SEND(300,400,2)
```

The above example calls the user-defined macro SEND by assigning the values 100, 200, 3 and 300, 400, 2 to the parameters FROM, TO, and COUNT respectively. The first call transfers the three data values from address 100 to follow address 200. The second call transfers the two data values from address 300 to follow address 400. The MP macroprocessor analyzes the macro calls coded in source text and performs text expansion, as shown below.

```
        MOV     X1,#0
        MOVB    R2,#3
ACT00:
        MOVB    A,100[X1]
        MOVB    200[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT00
        MOV     X1,#0
        MOVB    R2,#2
ACT01:
        MOVB    A,300[X1]
        MOVB    400[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT01
```

1.2 Processor Overview

When MP is invoked, it first evaluates the invoking command. If the command line contains a command format error, then MP will output a command error message and return to the operating system. If there is no error, then MP accepts the invoking command and moves on to text processing.

During text processing, MP checks for I/O errors related to specified file existence, memory capacity and disk capacity, as well as syntax errors in macro definitions and macro calls.

If an I/O error is detected, then MP will assume it is a fatal error. After it outputs an error message to the console, it will return to the operating system.

If a syntax error is detected, then MP will recognize it as a macro error. It will output an error message to the console, but processing will continue. However, text at the location of detected syntax errors will not be expanded. The original source text contents will be output instead.

When MP ends text processing with no command format errors or fatal errors, it outputs an expanded file. In addition, an output file for error messages can be specified as an option when MP is invoked.

Figure 1-2 shows the above process flow.

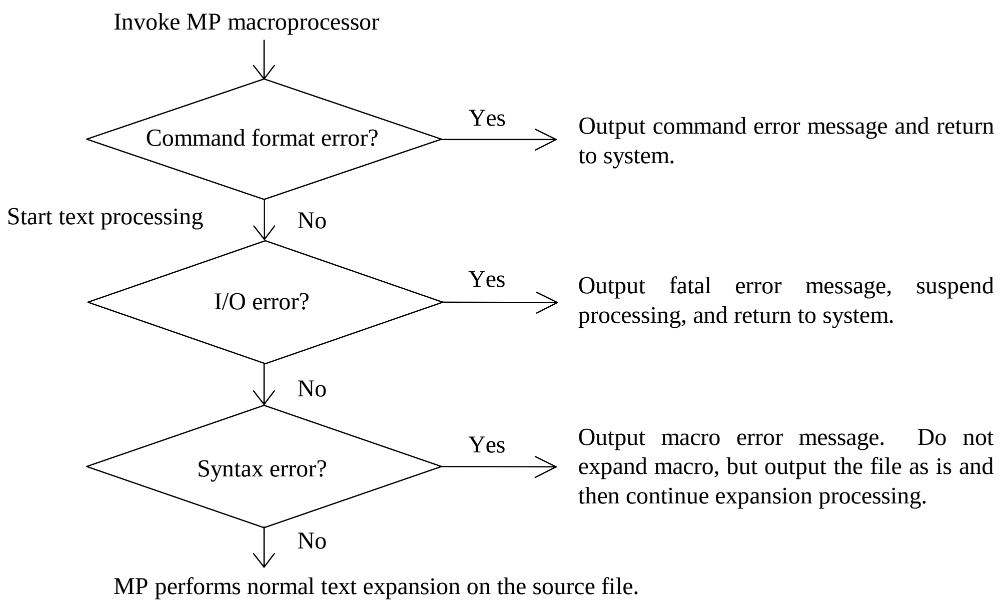

Invoke MP macroprocessor
Command format error?
Yes
Output command error message and return to system.
Start text processing
No
I/O error?
Yes
Output fatal error message, suspend processing, and return to system.
No
Syntax error?
Yes
Output macro error message. Do not expand macro, but output the file as is and then continue expansion processing.
No
MP performs normal text expansion on the source file.

Figure 1-2. Process Flow After Invoking MP

Please refer to Section 2, “MP Operation,” for further details about invoking MP. Refer to Section 7, “Error Messages,” regarding error messages.

1.3 Macroprocessor Language

The basic functions of MP are string substitutions. These functions can be used according to certain predetermined methods. Macroprocessor language defines these methods. In other words, macroprocessor language defines the formats for calling the functions of MP.

MP functions can be used by calling macros. MP provides several predefined macro functions, known as standard macros.

In general, if a macro is not defined in advance, then it cannot be called. However, the standard macros are already defined, so they can be used just by calling them. Standard macros cannot be redefined within the text.

All macro calls are coded following an @ symbol (called the metacharacter). MP reads through text searching for the metacharacter. When a metacharacter is encountered, it evaluates the following character string as a macro, and replaces the macro call with a return value.

The metacharacter can be changed to a different character with the standard macro METACHAR.

1.4 MP Output

MP outputs three types of files. The output destinations for these files are predetermined, but they can be changed when MP is invoked.

File Type	Decription
Expanded file	This file contains the text after macro expansion. Expanded text is normally output to disk.
Symbol file	This file is a listing of user macros. This file is created only when the SPA or SPB option has been specified.
Error file	This file lists the error messages generated during macro processing. Error messages are normally output to the console. An output file can be specified with the EP option.

If the path name is omitted for an output file specification, then the output file will be created in the current directory. If only the path name is specified, the output file will be given the same name as the source file, and it will be given the default extension appropriate for its type. Table 1-2 shows the default extension for each output file.

Table 1-2. Output File Default Extensions

File Type	Default File Extension
Expanded file	.Q
Symbol file	.S
Error file	.E

1.5 Path and File Specifications

Path names and file names can be specified on the invoking command line and in several parts of the text. In the explanation below, the parts of specification for path name or file name will be declared by convention as *path specification* and *file specification* respectively.

In general, a file name is indicated with the following format.

```
[ drive name : ][\][ directory name \ ]...[ directory name \ ] file name [ .file extension ]
|────────────── path specification ─────────────|
|────────────── file specification ─────────────────────────────────────|
```

The above example shows the ranges for *path specification* and *file specification*.

A *path specification* is specified as a drive name and/or directory path name. Note that directory names must end with the backslash (\). Below are some examples of directory declarations.

A : \	(specifies the root directory of the A drive)
A : \USR\PART1\	(specifies the \USR\PART1 directory of the A drive)
\USR\PART1\	(specifies the \USR\PART1 directory of the current drive)

A *file specification* is specified in the order of drive name, path name, file name, and its extension.

1.6 File Inclusion

MP provides a function for incorporating the contents of at any position in the text while it is running. This function can be easily used by calling a file include macro (INCLUDE or MACROLIB).

File include macros can be nested up to 13 levels, allowing other files to be included within an included file.

1.7 Code Examples

In order to explain the macroprocessor’s basic functions, this manual uses some macro code examples that may not completely follow assembler syntax.

2. MP Operation

This section explains how to operate MP.

2.1 Invoking MP

MP is invoked by typing the following at the operating system prompt.

MP *file specification* [*options*]

The *file specification* specifies just one source file to be processed.

The *options* can be used to specify just the necessary options (explained later) in any order. When multiple options are specified, they are delimited by spaces. Refer to Table 2-1, “Option Table,” for further details on the options.

In the invoking command, “MP,” *file specification* and *options* are delimited by spaces.

If MP is invoked only with an invoking command, then that command line will be shown on the console.

2.2 Source File Name

If the source file name is specified with an extension appended to a file name, then MP will recognize that specification as the source file name. If the extension is not appended, then MP will assume an extension “.ASM.” However, if the file name ends with a period (.), then MP will assume a file name with no extension.

If a drive name and path name are not specified, then MP assumes that the file is in the current directory.

Below are some examples of extension specifications and interpretations.

Specification	Interpretation
TEXT.SRC	TEXT.SRC
TEXT	TEXT.ASM
TEXT.	TEXT

2.3 Options

The processes performed by MP are already determined, but many of them can be optionally selected. Selection of processes is role of options.

Abbreviated forms are available for several options in order to shorten expressions. The same result is obtained no matter which form is used. For example, both of the following two options give the same result.

```
ERRORPRINT(OUT.ERR)
EP(OUT.ERR)
```

In general, the same option cannot be used multiple times in a single command line. However, the DEF option only can be used any number of times in a single command line.

Table 2-1. Option Table

Option Format	Abbreviation	Function
ERRORPRINT[(*file specification*)]	EP	Specifies output destination of error messages.
EXT[(*file specification*)]	-	Specifies output destination of text after macroexpansion.
SPA [(*file specification*)]	-	Specifies output destination of symbol table contents.
SPB[(*file specification*)]	-	Specifies output destination of symbol table contents.
INCLUDE(*path specification*)	IC	Specifies path name of include files.
MACROLIB(*path specification*)	ML	Specifies path name of include files.
GEN[(*mark specification*)]	GE	Specifies format of output text.
GENONLY	GO	Specifies output of expanded lines only as output text.
DEFINE(*symbol*)(*string*)	DEF	Defines user symbols.
DL	-	Deletes white space generated by macro expansion.

Note: '-' indicates that no abbreviated form exists.

When MP is invoked, the following options are specified as defaults.

Option	Function
ERRORPRINT	Outputs to console.
EXT	Outputs to a file with the source file name and the extension ".Q."
GENONLY	Outputs only expanded lines.

Each of the options is described below.

- **ERRORPRINT[(*file specification*)]** **Abbreviated form: EP**

This option specifies the output destination of error messages. Error messages are normally output to the console, but they can be directed to a file or a printer by using this option.

If the file specification is omitted, then the error file name will be the source file name with its extension replaced by ".E."

● **EXT[(*file specification*)]** **Abbreviated form: None**

This option specifies the output destination of the expanded text. Expanded text is normally output to a disk file given the source file name with its extension replaced by ".Q." but it can be directed to a different file or a printer by using this option.

If the file specification is omitted, then the expanded text file name will be the source file name with its extension replaced by ".Q."

● **SPA[(*file specification*)]** **Abbreviated form: None**
● **SPB[(*file specification*)]** **Abbreviated form: None**

These options specify the creation and output of a symbol file. A symbol file is normally not created, but a listing can be generated and output to a file or printer by using these options.

If the file specification is omitted, then the symbol file name will be the source file name with its extension replaced by ".S."

There are two types of symbol lists, which can be selected using SPA and SPB respectively. The difference between the two lists is that one includes user macro contents and the other does not. User macro contents will be listed if SPA is specified, but will not be listed if SPB is specified.

Symbols are output in alphabetical order. Each symbol's defined contents (string) is shown. If a symbol is a macro name, then the string "======USER MACRO======" will be shown, followed by its code format, local label, and macro body.

When SPA is specified, MP will create a listing in the format below.

```
MP Macro Processor, Ver. 1.10 Symbol List
Date : 1989-01-30 (Mon) 14:16:09

  Symbol            String

  BaseVAL.........  100H
  DebugFLG........  1
  GETP............  ======= USER MACRO =======
           Format  : GETP ( DX1 , DX2 )
           Body    :
                        MOVB    R0,#@DX1
                        MOVB    R1,#@DX2
  SEND............  ======= USER MACRO =======
           Format  : SEND ( FROM , TO , COUNT )
           Label   : ACT
           Body    :
                        MOV     X1,#0
                        MOVB    R2,#@COUNT
                    @ACT:
                        MOVB    A,@FROM[X1]
                        MOVB    @TO[X1] ,A
                        INC     X1
                        DECB    R2
                        JC      NE,@ACT
```

When SPB is specified, MP will create a listing in the format below.

```
MP Macro Processor, Ver. 1.13 Symbol List
Date: 1989-01-30 (Mon)  14:16:09

  Symbol              String
  BaseVAL.........    100H
  DebugFLG........    1
  GETP............    ======= USER MACRO =======
  SEND............    ======= USER MACRO =======
```

● **INCLUDE[(*path specification*)]** **Abbreviated form: IC**

This option is used to forcibly set the path name of files specified by INCLUDE macros within the text. This option affects all INCLUDE macros within the text.

For example, assume the following text is in source file FILE.ASM.

```
@INCLUDE(A:\TOOL\PACK\RAM1.DAT)            ; ......①
@INCLUDE(B:\MACDEF\RAM2.DAT)               ; ......②

        MOV     A,ER0
        MOV     ERI,#3000H
        ADD     A,ERI
          :
          :
```

Now assume that the MP macroprocessor is invoked with the following command line.

```
MP FILE.ASM IC(C:\ROOT\)
```

In this case, the include files specified in the source text will be taken as those below.

```
C:\ROOT\RAM1.DAT        ......①
C:\ROOT\RAM2.DAT        ......②
```

● **MACROLIB[(*path specification*)]** **Abbreviated form: ML**

This option is used to forcibly set the path name of files that are specified by MACROLIB macros within the text. This option affects all MACROLIB macros within the text.

For example, assume the following code in the text.

```
@MACROLIB(A:\TOOL\PACK\RAM1.DAT)
```

Now assume that ML(\ROOT\) is specified when MP is invoked. The include file will then be handled as follows.

```
\ROOT\RAM1.DAT
```

● **GEN[(*mark specification*)]** **Abbreviated form: GE**
● **GENONLY** **Abbreviated form: GO**

These options specify the output format of expanded text.

GENONLY specifies that only expanded text is to be output.

GEN specifies that both expanded text and all macro definitions are to be output. The string given as the mark specification is to be inserted before each macro definition. If the mark specification is omitted, then ";++" will be the default mark. This allows the macro definitions to be processed as comments when MP output is used as assembler source.

These options are effective until a GEN or GENONLY macro is called within the text. The default option is GENONLY. Some expansion examples are shown below:

Text before expansion:

```
@MACRO(GETP(DX1,DX2))(
        MOVB    R0,#@DX1
        MOVB    R1,#@DX2
)

        MOV     A,#50H
@GETP(10H,20H)

        RT
```

Text after expansion (when GENONLY is specified):

```
        MOV     A,#50H
        MOVB    R0,#10H
        MOVB    R1,#20H

        RT
```

Text after expansion (when GEN is specified):

```
;++@MACRO(GETP(DX1,DX2))(
;++     MOVB    R0,#@DX1
;++     MOVB    R0,#@DX2
;++)

        MOV     A,#50H
;++@GETP(10H,20H)
        MOVB    R0,#10H
        MOVB    R1,#20H

        RT
```

Text after expansion (when GEN(;) is specified):

```
;@MACRO(GETP(DX1,DX2))(
;       MOVB    R0,#@DX1
;       MOVB    R1,#@DX2
;)

      MOV       A,#50H
;@GETP(10H,20H)
        MOVB    R0,#10H
        MOVB    R1,#20H

        RT
```

● DL **Abbreviated form: None**

This option prevents any white space created by macro expansion (lines of spaces, tabs, and carriage return codes only) from being output to the expanded file. White space that exists in the original source file will be output as is.

Source text:

```
@MACRO(GETP(DX1,DX2))(
        MOVB    R0,#@DX1
        MOVB    R1,#@DX2
)

        MOV     A,#50H
@GETP(10H,20H)

        RT
```

Expanded text (DL option specified):

```
        MOV     A,#50H
        MOVB    R0,#10H
        MOVB    R1,#20H

        RT
```

Expanded text (DL option not specified):

```
        MOV     A,#50H

        MOVB    R0,#10H
        MOVB    R1,#20H

        RT
```

Unless otherwise stated, expanded text shown in this manual is always the result obtained by specifying the DL option.

● DEFINE (*symbol*) (*string*) **Abbreviated form: DEF**

This option defines a symbol as having a particular string as its value. Defined symbols can be referenced within the source text as user symbols. Examples of DEFINE option use are given below.

The macro symbol VAR is called in the following source text. Assume VAR is not defined as a macro within the text. If MP processes the text as is, then an undefined error will occur.

```
        MOV     A,#@VAR
```

The symbol VAR is defined when MP is invoked by using the DEFINE option.

```
MP TEXT.ASM DEFINE(VAR)(10H)
```

As a result, VAR is evaluated as 10H. The text is expanded as follows.

```
        MOV     A,#10H
```

Simple string replacement is not all that is possible. Text expansion can be controlled by combining expansion control macros and the DEFINE option. By using the expansion control macro IFDEF in the following text, expansion is controlled by whether or not the symbol SYM has been defined.

```
@IFDEF(SYM)THEN(
        ADD     A,ER0
)ELSE(
        MOV     ER0,A
)FI
```

The symbol SYM is defined when MP is invoked by using the DEFINE option.

```
MP TEXT.ASM DEFINE(SYM)(1)
```

This will expand the text as follows.

```
        ADD     A,ER0
```

Conversely, if the symbol SYM is not defined when MP is invoked, then the text will be expanded as follows.

```
        MOV     ER0,A
```

2.4 Screen Display

Between its start and end, MP displays several types of information on the console. These are described here through an example.

The following figure is one example of MP's screen display from start to finish.

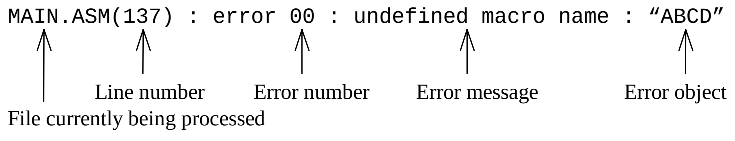
```
   A>MP \USR\MYDIR\MAIN EXT(\USR\) EP(LST)
   MP Macro Processor, Ver.1.10 Jan 1989
   Copyright (C) 1988,1989. OKI Electric Ind. Co.,Ltd.

①  MAIN.ASM(137) : error 00 : undefined macro name : “ABCD”

②  1 Macro Errors Found
   A>
```

When an error occurs, the error line and a message are displayed on the console. Number 1 at the far left of the figure corresponds to such a line. It is displayed in the following format

```
MAIN.ASM(137) : error 00 : undefined macro name : “ABCD”
```



Note that if an error occurs in a macro that spans multiple lines, then the first line of that macro will be displayed.

```
001    @SET(VAL,5)
002    @WHILE(@VAL)(       <———————————————————— Line number in message display
003            MOV     A,@UNDEF_MACRO        <—— Line number actually causing the error
004            @SET(VAL,@VAL-1)
005   )
 ↑
 └────────── Line number
```

Number 2 above indicates MP’s termination message. In this example it indicates that there was one error. If there are no errors, the following message will be displayed.

```
Macro Error Not Found
```

3. Macroprocessor Language Format

This section explains the rules and source text syntax of macroprocessor language.

3.1 Character Set Usable in Source Text

All characters expressed with one byte can be used.

3.2 Macro Definitions, Calls, and Expansion

The creation of user-defined macros through the use of the standard macro MACRO is referred to as "macro definition" by this manual. Coding a standard macro or previously defined user-defined macro within the source text is referred to as a "macro call." MP's process of analyzing a called macro and expanding the text is referred to as "macro expansion."

MP does not make forward references. Accordingly, before a macro can be called it must have a corresponding macro definition in the source text or it must be a standard macro provided by MP.

MP recognizes and expands both standard macros provided by MP and user-defined macros provided by the user. The parameters and format of standard macros must follow pre-determined rules. For details, refer to Section 5, "Standard Macros."

User-defined macros are macros for which the user defines a user symbol, parameters, and a coding format through the use of the standard macro MACRO. The user can define any symbols, parameters, and coding format for a macro, but the syntax of each call to that macro must correctly match the definition. A user-defined macro not previously defined in the source text cannot be called.

Figure 3-1 shows the relation between macro definitions, macro calls, and macro expansion.

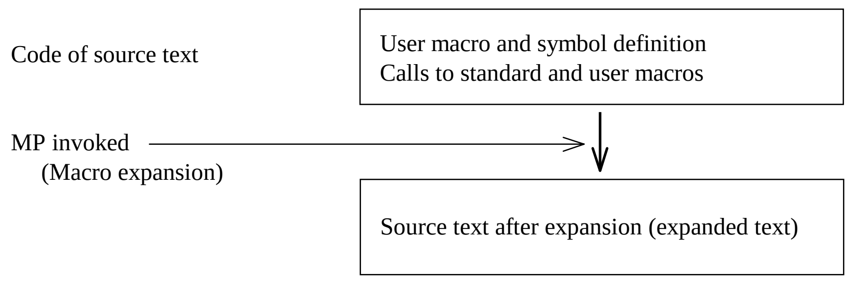

Code of source text
User macro and symbol definition
Calls to standard and user macros
MP invoked
(Macro expansion)
Source text after expansion (expanded text)

Figure 3-1. Concept of Macro Expansion Process

For specific examples of the process shown above, refer to Section 4, "Simple Macro Examples."

3.3 Symbols

MP defines symbols as follows.

Characters usable in symbols are those below.

A–Z   a–z   0–9   ?   _ (underscore)

However, in order to distinguish symbols from numbers, the first character of a symbol must not be a digit.

Symbol length is unlimited, but only the first 31 characters are valid. Characters beyond 31 are ignored.

The following items are all handled as symbols.

(1) Standard macro names
(2) Keywords used in some standard macros (LOCAL, THEN, ELSE, FI)
(3) User symbols
(4) User macro names
(5) User macro parameters
(6) User macro local labels
(7) Some operators (refer to section 3.4)

MP distinguishes between upper-case and lower-case letters in (3)-(6) above. For example, ‘TELEX’ and ‘telex’ are handled as separate symbols.

MP does not distinguish between upper-case and lower-case letters for the others. For example, ‘MACRO’ and ‘macro’ are handled as the same symbol.

Only standard macro names (1) cannot be used as user symbols, user macro names, parameters, or local labels.

3.4 Expression Format

Several standard macros take expressions as parameters. These macros recognize and operate on the character strings placed at those parameters as expressions. Non-expression parameters are all evaluated as simple character strings.

MP handles numbers internally as signed 32-bit numbers

Expressions are constructed by combining constants and operators, but it is also possible to include a macro call within an expression.

Any number of spaces, tabs, or carriage returns may be included between the constants and operators in an expression. This white space does not have any affect on the calculation.

Expression format rules are described below.

3.4.1 Constants

MP handles strings that start with a digit 0–9 as constants. Constants can be expressed as binary, octal, decimal, or hexadecimal numbers. In order to differentiate between these forms, a type descriptor is appended to the number, as shown in the table below. The type descriptor may be omitted only for decimal numbers.

If the first character of a hexadecimal number would be alphabetic (A-F), then it must be prefixed with the digit 0 to distinguish it from a symbol. For example, the hexadecimal number AH would be prefixed with 0 to be coded 0AH.

Constants may use lower-case alphabetic letters.

Radix	Usable characters	Type descriptor	Code examples
2	0, 1	B	1010B,01101101B
8	0–7	O, Q	271O, 514Q
10	0–9	D	30D, 1263
16	0–9, A–F	H	753H, 0C6E7H

3.4.2 Operators

Operators are of two types, those expressed with alphabetic characters and those expressed with special characters. When using a operator expressed with alphabetic characters, it is delimited from other characters by placing one or more spaces, tabs, or carriage return codes before and after the operator.

3.4.2.1 Arithmetic operators

Operator	Function
+	Addition or unary positive operator.
–	Subtraction or unary negative operator.
*	Multiplication.
/	Division.
%	Modulo calculation (returns remainder of the left term divided by the right term).

■ **Examples** ■

Expressions that use these operators and their values are shown below.

Expression	Value
1234H + 80H	12B4H
1234H – 80H	11B4H
1234H * 80H	1A00H
1234H / 80H	24H
1234H % 80H	34H

3.4.2.2 Logical operator

The results of these operations are always 0 (false) or –1 (true).

Operator	Function
!	Retruns –1 if right term is zero. Otherwise returns 0.

■ **Examples** ■

Expressions that use this operator and its value is shown below.

Expression	Value
! 5588H	0

3.4.2.3 Bitwise logical operators

These operators perform operations on each bit of operational term.

Operator	Function
~	Bit inversion of right term.
&	Logical AND of left and right terms.
|	Logical OR of left and right terms.
^	Exclusive OR of left and right terms.
<<	Shifts left term to the left by the number of bits expressed by the right term. Zeroes are shifted in from the right (least significant bit).
>>	Shifts left term to the right by the number of bits expressed by the right term. Zeroes are shifted in from the left (most significant bit).

■ **Examples** ■

Expressions that use these operators and their values are shown below.

Expression	Value
1234H & 4321H	0220H
1234H | 4321H	5335H
1234H ^ 4321H	5115H
1234H << 1	2468H
1234H >> 1	091AH
~1234H	0EDCBH

3.4.2.4 Special operators

The results of these operations are always 1-byte values.

Operator	Function
HIGH	Returns high byte of the right term.
LOW	Returns low byte of the right term.

■ **Examples** ■

Expressions that use these operators and their values are shown below.

Expression	Value
HIGH 1234H	12H
LOW 1234H	34H

3.4.2.5 Relational operators

The results of these operations are always 0 (false) or -1 (true).

Operator	Function
>	Returns -1 if left term is greater than right term. Otherwise returns 0.
>=	Returns -1 if left term is greater than or equal to right term. Otherwise returns 0.
<	Returns -1 if left term is less than right term. Otherwise returns 0.
<=	Returns -1 if left term is less than or equal to right term. Otherwise returns 0.
==	Returns -1 if left term is equal to right term. Otherwise returns 0.
!=	Returns -1 if left term is not equal to right term. Otherwise returns 0.

■ **Examples** ■

Expressions that use these operators and their values are shown below.

Expression	Value
1234H > 1234H	0
1234H >= 1234H	-1
1234H < 1234H	0
1234H <= 1234H	-1
1234H == 1234H	-1
1234H ! > 1234H	0

3.4.2.6 Precedence

Operators are not evaluated in the sequence in which they are coded. They are evaluated in accordance with some predefined rules, known as precedence rules. The table below shows the precedence of operators. The highest precedence is 1, with lower precedence following larger numbers.

Operators on the same line have the same precedence. Operators are always evaluated in order from highest to lowest precedence. Operators with the same precedence are evaluated in the order that they are coded, starting from the left of the expression.

Precedence	Operators
1	( )
2	! ~ –(unary) HIGH LOW
3	* / %
4	+ –
5	<< >>
6	< <= > >=
7	== !=
8	&
9	^
10	|

3.5 Balanced Text

Most standard macros require parameters when called. Parameters must be enclosed in left and right parentheses ( ). Parentheses can also be placed between parentheses, but the left and right parentheses must balance. In other words, a string that starts with a left parenthesis ‘(’ must close with a corresponding right parenthesis ‘)’. Parentheses without meaning cannot be included within a string. However, nesting of parentheses as described above is allowed.

A character string contained in balancing left and right parentheses is referred to as “balanced text.” There is no limit on the number of characters contained within a balanced text. Null strings are also permitted. Balanced text may also include any desired macro calls. Standard macros are processed as balanced text for which all calls within have been expanded.

The parameter list, local list, and macro body of the MACRO macro, as well as expressions used as macro parameters, occupy a special position within balanced text. Such text can also include any desired macro calls.

The examples below show balanced and unbalanced text.

[Balanced text example]

```
@IF(@EQS(@OPERATION,SUB)) THEN(
        SUB    A,ER0
)FI
```

[Unbalanced text example]

```
@IF(@EQS(@OPERATION,SUB THEN(
        SUB    A,ER0
)FI
```

In the above text, the numbers of left and right parentheses do not balance. When MP encounters unbalanced text, it outputs a macro error message to the console and does not perform text expansion on that portion of text.

4. Simple Macro Examples

4.1 MACRO Macro Examples

The standard macro MACRO is used to create user-defined macros. The following examples explain macro definitions, calls, and expansion. For details on user macro parameters and local labels, refer to Section 5.1, "User-Defined Macros."

Shown below is the simplest macro definition.

```
@MACRO(GETP)(
        MOVB    R0,#10H
        MOVB    R1,#20H
)
```

This example is the macro definition for the macro named GETP. This macro is called as follows.

```
@GETP
```

The resulting output text will be expanded as follows.

```
        MOVB    R0,#10H
        MOVB    R1,#20H
```

However, this macro always expands into the same contents, and cannot be used for other purposes. It can be made more general purpose by using parameters in its definition, as shown below.

```
@MACRO(GETP(ADR1,ADR2))(
        MOVB    R0,#@ADR1
        MOVB    R1,#@ADR2
)
```

The macro defined here is named GETP and has two parameters (ADR1, ADR2).

When the macro is called, the parameters will be replaced with the real arguments for that call. When parameters are referenced in a macro body, a metacharacter must be attached before the first character. In the above example, the parameters ADR1 and ADR2 are used within the definition of the user macro GETP(ADR1,ADR2). Within the macro body inside the following parentheses, these parameters are referenced by prefixing them with metacharacter (here @), such as @ADR1 and @ADR2.

A macro defined in this way is called as follows.

```
@GETP(10H,20H)
```

The resulting output text is expanded as follows.

```
        MOVB    R0,#10H
        MOVB    R1,#20H
```

Next, an example of a macro that uses local labels will be shown. Local labels are labels that are valid only within their own macro body. In the example below, ACT is a local label. Local labels must also be prefixed with the metacharacter when referenced.

```
@MACRO(SEND(FROM,TO,COUNT)) LOCAL ACT(
        MOV     X1,#0
        MOVB    R2,#@COUNT
@ACT:
        MOVB    A,@FROM[X1]
        MOVB    @TO[X1],A
        INC     X1
        DECB    R2
        JC      NE,@ACT
)
```

This macro could be called as follows.

```
@SEND(100,200,3)
@SEND(300,400,2)
```

The resulting output text is expanded as follows. Local labels in the expanded macros are kept unique by automatically appending two or four digits to them in accordance with their order of reference.

```
        MOV     X1,#0
        MOVB    R2,#3
ACT00:
        MOVB    A,100[X1]
        MOVB    200[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT00
        MOV     X1,#0
        MOVB    R2,#2
ACT01:
        MOVB    A,300[X1]
        MOVB    400[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT01
```

4.2 DEFINE Macro Examples

Within the standard macro library, the DEFINE, SET, and MATCH macros have similar functions to the MACRO macro. However, these macros simply assign strings to symbols, while symbols defined with the MACRO macro can use parameters and local labels.

The most different point is that the MACRO macro only defines user macros. It does not perform expansion of the macro body, which codes the macro’s functions, until the defined macro name is called.

Conversely, the DEFINE, SET, and MATCH macros expand symbols to the text at the point where the symbols are defined. Accordingly, the values at the time of macro definition are used wherever the macro is to be called.

The following examples show a DEFINE macro definition and a MACRO macro definition, and explain the differences.

[DEFINE definition example]

```
@DEFINE(VALUE)(1)
@DEFINE(CAT)(@VALUE)
@CAT
@DEFINE(VALUE)(2)
@CAT
@VALUE
```

In the above example, the symbol VALUE is defined as the string ‘1’ in the first line. The symbol CAT is assigned the value of VALUE in the second line. Therefore, the value of CAT when called in the third line becomes ‘1.’ Then the value of VALUE is redefined in the fourth line. However, the value of CAT was already determined in the second line, so changing the value of VALUE to 2 does not affect the value of CAT. Therefore, the value of CAT when called in the fifth line becomes ‘1.’ Of course the value of VALUE in the sixth line is 2.

Next is an example of a MACRO macro definition.

[MACRO definition example]

```
@DEFINE(VALUE)(1)
@MACRO(CAT)(@VALUE)
@CAT
@DEFINE(VALUE)(2)
@CAT
@VALUE
```

In this example, the second line of the DEFINE definition example is rewritten as a MACRO definition. The defined user macro CAT is expanded when called, and a value is substituted for its parameter. Therefore, when CAT is called in the fifth line, the macro body of the second line (@VALUE) is expanded, and since VALUE was redefined in the fourth line, that new value will be used. In this example, VALUE was redefined as the string ‘2,’ so that becomes the expanded contents of the user macro call of CAT in the fifth line.

Thus, the result of expanding a user macro defined by MACRO can change depending on the status at the time of the call. The contents of a user macro defined by DEFINE (or SET or MATCH) are determined at the time of definition, so the expanded contents will always be the same unless it is redefined.

Hereafter this manual refers to items defined with the MACRO macro as user-defined macros or user macros, and items defined with the DEFINE macro as user-defined symbols or user symbols. Both are strictly distinct.

5. Standard Macros

This section explains MP's standard macros in detail. Standard macros are classified as shown in Table 5-1.

Table 5-1. Standard Macro Types

Type	Macro name
User macro definition	MACRO
Symbol definition	DEFINE, SET, MATCH
Process control	Comment, Bracket, Escape, METACHAR
String comparison	EQS, NES, LTS, LES, GTS, GES
String operation	EVAL, LEN, SUBSTR
Expansion control	IF, IFDEF, IFUNDEF, WHILE, REPEAT, EXIT
Console I/O control	IN, OUT, COLOR
File include	INCLUDE, MACROLIB
Listing control	GEN, GENONLY
System control	SYSTEM, EXIST, SOURCE, FDATE, FTIME, DATE, TIME
Symbol definition purging	PURGE, ALLPURGE

Each of the following sections split their descriptions into the items below.

■ **Format** ■	Format for calling the standard macro.
■ **Function** ■	The macro's function and purpose.
■ **Examples** ■	Specific example of the macro's use.

However, Section 5.1 "User-Defined Macros" explains not only how user macros are defined with the MACRO macro, but also how user macros are called.

5.1 User-Defined Macros

5.1.1 MACRO

■ **Format** ■

@MACRO (*macro_name* [*parameter_list*]) [*local_list*] (*macro_body*)

■ **Function** ■

MACRO is provided for the user to create macros with proprietary functions. Macros defined with MACRO are called user macros. The MACRO call itself is expanded to a null string.

The *macro_name* is the name that will identify the defined macro. It is used when the macro is called.

The *macro_body* encodes the functions of the macro. It is the value returned by a call to the macro. The macro body must be balanced text enclosed within parentheses.

The simplest macro definition consists of only a macro name and a macro body. However, these alone do not allow macros to be used effectively. By adding parameter_lists and local_lists, macros with more complex functions can be defined.

Note: Where confusion is possible in the explanation below, parameter symbols coded in user macro definitions are referred to as “dummy parameters,” and parameters passed when macros are called are referred to as “real parameters.”

The *parameter_list* lists the symbols that are used as parameters (dummy parameters) in the macro body. When the macro is called, these parameters accept the arguments that are passed (real parameters). All parameters used within the macro body must be listed here. If a parameter is not used, then it should be omitted from the parameter list. Parameter list syntax is as follows.

delimiter symbol [[delimiter symbol] ...] delimiter

Any single character other than those shown below can be a delimiter. However, if parentheses are used, then left and right parentheses must balance.

Characters not allowed as delimiters:

A–Z, a–z, 0–9, _ (underscore), ?, currently valid metacharacter

To reference a parameter within the macro body, prefix it with the metacharacter.

The rules for specifying a parameter list in macro definitions and calls are described through the use of some simple examples below. First, some macro definitions are shown.

```
@MACRO(UPRG1(P1,P2,P3))(            .....①
        MOV     A,#@P1+@P2+@P3
)
@MACRO(UPRG2(P1,P2,P3))(            .....②
        MOV     A,#@P1+@P2+@P3
)
@MACRO(UPRG3 P1,P2,P3 )(            .....③
        MOV     A,#@P1+@P2+@P3
)
```

Example ① is a macro definition that uses parentheses and commas as delimiters. Example ② is a macro definition that uses parentheses and spaces as delimiters. Example ③ is a macro definition that uses spaces for all delimiters. The space to the right of parameter P3 could be omitted.

The format of the parameter list for a macro call must exactly match that of the definition. Accordingly, calls corresponding to the above three definition examples are performed as follows.

```
   @UPRG1(10H,20H,30H)
   @UPRG2(10H 20H 30H)
   @UPRG3 10H 20H 30H
```

These macro calls will be expanded as follows.

```
            MOV     A,#10H+20H+30H
            MOV     A,#10H+20H+30H
            MOV     A,#10H+20H+30H
```

The *local_list* lists the symbols that are used as local labels in the macro body. It follows the keyword ‘LOCAL.’ If a local list is not needed, then it can be omitted. Local list syntax is as follows. The delta symbol(Δ) can be spaces, tabs, or carriage return codes.

LOCAL Δ *symbol* [Δ *symbol*...]

The programmer may want some symbols to be expanded into different symbols each time the macro is expanded. For example, such symbols include labels that are valid only within the macro body. Symbols used as label definitions must be expanded into different symbols each time their macro is expanded. Otherwise a double definition error will occur during assembly. If these symbols are declared as local labels in the macro definition, then they will generate unique symbols each time the macro is called.

Local labels are created by the following procedure. MP has an internal counter for local labels. Each time a macro that includes local labels is called, the counter is incremented by one. The counter starts at 0 and counts to FFFFH. After FFFFH it returns to 0 and repeats its counting. The value of this counter, expressed in hexadecimal, is appended to the local labels. If the counter value is 0–FFH, then a 2–digit hexadecimal expression will be appended. For greater values, a 4–digit hexadecimal expression will be appended. In both cases, when the value itself does not fill all the digits then the leftmost digits will be filled with zeroes.

Symbols used in parameter lists and local lists:

- There is no limit to the number of symbols. However, the parameters and local labels in a user macro must all be different symbols.

- Symbols in parameter lists and local lists may be the same as user macro names or general symbol names. These list symbols are valid only within the macro body.

- The contents of parameters and local labels cannot be modified within the macro body. If a same name is redefined with DEFINE, then it will be handled not as a parameter or local label, but as a symbol definition.

  If a user macro name or user symbol exists with the same name as a dummy parameter or local label, then it will be interpreted as the dummy parameter or local label during user macro expansion.

Below is an example of a macro that uses local labels. In this example ‘LOOP’ and ‘LEND’ are declared as local labels.

```
@MACRO(TCMP)LOCAL LOOP LEND(
@LOOP:
        CAL     getTime
        DECB    R2
        JC      NE,@LEND
        CAL     cmpTime
        DECB    R2
        JC      NE,@LEND
        J       @LOOP
@LEND:
)
```

Assume the macro is called as follows.

```
@TCMP
@TCMP
```

As a result, the macros are expanded as follows. Each time the macro is expanded, ‘LOOP’ and ‘LEND’ are generated as different symbols.

```
LOOP00:
        CAL     getTime
        DECB    R2
        JC      NE,LEND01
        CAL     cmpTime
        DECB    R2
        JC      NE,LEND01
        J       LOOP00
LEND01:
LOOP02:
        CAL     getTime
        DECB    R2
        JC      NE,LEND03
        CAL     cmpTime
        DECB    R2
        JC      NE,LEND03
        J       LOOP02
LEND03:
```

Local labels are used to generate symbols that are valid locally. “Local” means that a symbol exists only in the expansion level that includes that symbol. In other words, the symbol cannot be directly referenced from higher or lower levels.

A macro body is first expanded when a currently defined user macro is called. Thus, even if an error occurs in the coding of a macro body, the actual error will be generated not when the macro is defined but when it is called.

MACRO can be called again within a macro body. In effect, one can create a user macro that defines user macros. In such cases, the inner macro will be defined at the point that the outer macro is called. The parameters and local labels of the outer macro are invalid in the inner macro body.

Another user macro can be called from within a macro body. A macro can even call itself (recursive call). However, MP does not control infinite loops, so exercise such macro nesting with caution.

5.1.2 Calling User Macros

■ **Format** ■

```
@macro_name [real_parameter_list]
```

A user macro must be defined prior to being called. Also, the format of the parameter list when the macro is called (delimiters, parameter count) must be identical to the format when the macro was defined. However, a real parameter may be a null string.

If spaces, tabs, or carriage return codes were the delimiters during definition, then spaces, tabs, and carriage return codes between real parameters are ignored. If the delimiters during definition were not spaces, tabs, or carriage return codes, then all spaces, tabs, or carriage return codes before or after the delimiters will be handled as parts of real parameters.

When a user macro is called and its macro body is expanded, the real parameters will replace their corresponding dummy parameters. Also during expansion 2-digit or 4-digit numbers (hexadecimal expressions) will be appended to local labels. The numbers appended to local labels will run from 00 to FFFFH following the order of macro calls. As explained in the section on local lists, MP does not provide this counter for each macro, but rather MP has a single internal counter. Thus, the same number will not be appended to the same local label within a single expanded file. However, if the counter value exceeds FFFFH, then the counter value will return to 00.

Even if the metacharacter is different during a user macro's definition and call, the user macro will be valid. If the METACHAR macro is called within a user macro, then the newly defined metacharacter will be valid only within that user macro.

5.1.3 Examples of Defining and Calling User Macros

■ **Example 1** ■

Before expansion:

```
@MACRO(GETP)(
        MOVB    R0,#10H
        MOVB    R1,#20H
)
@GETP
```

After expansion:

```
        MOVB    R0,#10H
        MOVB    R1,#20H
```

■ Example 2 ■

This example uses brackets [] and commas ‘,’ as parameter delimiters.

Before expansion:

```
@MACRO(GETP[ ADR1, ADR2 ])(
        MOVB    R0,#@ADR1
        MOVB    R1,#@ADR2
)
@GETP[10H,20H]
```

After expansion:

```
        MOVB    R0,#10H
        MOVB    R1,#20H
```

■ Example 3 ■

This example uses space ‘,’ as parameter delimiters.

Before expansion:

```
@MACRO(GETP ADR1 ADR2 )(
        MOVB    R0,#@ADR1
        MOVB    R1,#@ADR2
)
@GETP 10H 20H
```

After expansion:

```
        MOVB    R0,#10H
        MOVB    R1,#20H
```

■ Example 4 ■

This example specifies parameter delimiters only between some parameters.

Before expansion:

```
@MACRO(GETP ADR1,ADR2 )(
        MOVB    R0,#@ADR1
        MOVB    R1,#@ADR2
)
@GETP 10H,20H
```

After expansion:

```
        MOVB    R0,#10H
        MOVB    R1,#20H
```

In this example, if white space follows the macro call’s real parameter delimiter ‘,’ then the second parameter will be taken as a null string.

■ Example 5 ■

This example uses local labels.

Before expansion:

```
@MACRO(SEND(FROM,TO,COUNT))LOCAL ACT(
        MOV     X1,#0
        MOVB    R2,#@COUNT
@ACT:
        MOVB    A,@FROM[X1]
        MOVB    @TO[X1],A
        INC     X1
        DECB    R2
        JC      NE,@ACT
)
@SEND(100,200,3)
@SEND(300,400,2)
```

After expansion:

```
        MOV     X1,#0
        MOVB    R2,#3
ACT00:
        MOVB    A,100[X1]
        MOVB    200[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT00
        MOV     X1,#0
        MOVB    R2,#2
ACT01:
        MOVB    A,300[X1]
        MOVB    400[X1],A
        INC     X1
        DECB    R2
        JC      NE,ACT01
```

■ Example 6 ■

This example shows locality of parameters and local labels.

Before expansion:

```
@SET(ADR,100H)
@DEFINE(LABEL)(EXAMPLE_TEXT)
@MACRO(TEST<ADR,REDEF>)LOCAL LABEL(
        IN_LABEL = @LABEL
        IN_ADR   = @ADR
        @DEFINE(REDEF) (NEW_TEXT)
        IN_REDEF = @REDEF
)
@TEST<200H,0FFFFH>
        OUT_LABEL = @LABEL
        OUT_ADR   = @ADR
        OUT_REDEF = @REDEF
```

After expansion:

```
          IN_LABEL = LABEL00
          IN_ADR   = 200H
          IN_REDEF = 0FFFFH
          OUT_LABEL = EXAMPLE_TEXT
          OUT_ADR   = 100H
          OUT_REDEF = NEW_TEXT
```

In this example, the user symbols ADR and LABEL have the same names as the parameter ADR and local label LABEL. The latter meanings are valid within the user macro. The values of the user symbols (ADR, LABEL) are passed without regard to the macro definition.

Also, a DEFINE macro is used within the user macro body. Even though it is meant to redefine a parameter (REDEF), REDEF is not a parameter so it will define REDEF as a user symbol from outside the macro.

■ Example 7 ■

This example shows a user macro called from within a macro. In this example, GETP and SEND are called from within SEND2. The parameters passed to SEND2 are further passed as parameters to these macros.

Before expansion:

```
   @MACRO(SEND2 DX1 DX2 DX3 )(
           @GETP @DX1 @DX2        @’defined in EX.3’
           ;
           @SEND(@DX1,@DX2,@DX3)  @’defined in EX.5’
   )
   @SEND2  100  200  3
```

After expansion:

```
           MOVB    R0,#100
           MOVB    R1,#200
           ;
           MOV     X1,#0
           MOVB    R2,#3
   ACT00:
           MOVB    A,100[X1]
           MOVB    200[X1],A
           INC     X1
           DECB    R2
           JC      NE,ACT00
```

■ Example 8 ■

This is an example of a recursive macro call. In this example, the RECTON macro will call itself until the parameter CNT is 0. Note that strings, not numbers, are passed as parameters. Any problems this causes can be solved by using the EVAL macro.

Before expansion:

```
@MACRO(RECTON(SRC,DST,CNT))(
        ;
        MOVB    A,@SRC            ; CNT=@CNT
        ADCB    A,@DST
        MOVB    @DST,A
        @IF(@CNT>0)THEN(
                @RECTON(@SRC+2,@DST+2,@CNT-1)
        )FI
)
@RECTON(20H,40H,3)
```

After expansion:

```
        ;
        MOVB    A,20H             ; CNT=3
        ADCB    A,40H
        MOVB    40H,A
        ;
        MOVB    A,20H+2           ; CNT=3-1
        ADCB    A,40H+2
        MOVB    40H+2,A
        ;
        MOVB    A,20H+2+2         ; CNT=3-1-1
        ADCB    A,40H+2+2
        MOVB    40H+2+2,A
        ;
        MOVB    A,20H+2+2+2       ; CNT=3-1-1-1
        ADCB    A,40H+2+2+2
        MOVB    40H+2+2+2,A
```

■ Example 9 ■

This is an example of self-redefinition. The macro redefines itself from within itself.

Before expansion:

```
@MACRO(SUB1)(
SUB:    MOVB    P0, #PORT0
        MOVB    P1, #PORT1
        DECB    R0
        RT
        ;
        @MACRO(SUB1)(
        CAL     SUB
        )
)
@SUB1
@SUB1
```

After expansion:

```
SUB:    MOVB    P0, #PORT0
        MOVB    P1, #PORT1
        DECB    R0
        RT
        ;
        CAL     SUB
```

■ **Example 10** ■

This example uses double expansion of macros and delimiters.

Before expansion:

```
@MACRO(ADD[P1+P2=P3])(
        @SET(@P1,@@P1+@P3)
)
@SET(CNT,10H)
CNT = @CNT
@ADD[CNT+=30H]
CNT = @CNT
```

After expansion:

```
CNT = 10H
CNT = 40H
```

In this example, the parameter P2 is not actually used. Thus, when the macro is called, it does not matter what is passed. This example passes a null string.

The '@@P1' in the macro body is expanded as follows.

@@P1 → @CNT → 10H

Therefore, the final form of the SET macro is as follows.

```
@SET(CNT,10H+30H)
```

By using MP's multiple expansion of macros in this way, user symbols can be redefined by passing them as parameters.

5.2 Symbol Definition

5.2.1 DEFINE

■ **Format** ■

@**DEFINE** (*symbol*) (*balanced_text*)
or
@**DEF** (*symbol*) (*balanced_text*)

■ **Function** ■

DEFINE assigns a character string expressed as balanced text to a symbol. The DEFINE call itself expands to a null string.

■ Example 1 ■

Before expansion:

```
@DEFINE(BYTELIST)(10H,20H,30H)
        DB      @BYTELIST
```

After expansion:

```
        DB      10H,20H,30H
```

■ Example 2 ■

Before expansion:

```
@SET(CAT,10H)
@DEFINE(SYM)(@CAT)
@SET(CAT,20H)
        MOVB    R0,#@SYM
@DEFINE(SYM)(@CAT)
        MOVB    R0,#@SYM
```

After expansion:

```
        MOVB    R0,#10H
        MOVB    R0,#20H
```

5.2.2 SET

■ Format ■

@**SET** (*symbol, expression*)

■ Function ■

SET assigns the value of an expression to a symbol. Symbols defined with SET can be redefined with another SET call or DEFINE call, and they can be redefined as user macros with the MACRO call.

■ Example 1 ■

Before expansion:

```
@SET(AA,1101B+10B)
@AA
@SET(BB,@AA << 2)
@BB
@SET(CC,(@AA+@BB)/2)
@CC
```

After expansion:

```
0FH
3CH
25H
```

■ **Example 2** ■

Before expansion:

```
@SET(COUNTER,3)
@WHILE(@COUNTER > 0)(
        DB      @COUNTER
        @SET(COUNTER,@COUNTER-1)
)
```

After expansion:

```
        DB      3H
        DB      2H
        DB      1H
```

5.2.3 MATCH

■ **Format** ■

@MATCH (*symbol* [[*delimiter symbol*]...]) (*balanced_text*)

■ **Function** ■

MATCH accepts balanced text as a character string, and assigns substrings as delimited by the delimiter to the symbols in order. The MATCH call itself expands to a null string.

The rules for delimiters are exactly the same as those for MACRO macro parameter delimiters.

If the number of symbols is fewer than the substrings, then the entire remainder of the string will be assigned to the final symbol.

■ **Example 1** ■

Before expansion:

```
@MATCH(AA|BB&CC)(DOG|CAT&PIG,MOUSE)
@AA
@BB
@CC
```

After expansion:

```
DOG
CAT
PIG,MOUSE
```

In this example, '|' and '&' are used as delimiters. For each delimiter found in the text, the corresponding symbol is defined.

■ **Example 2** ■

Before expansion:

```
@MATCH(W,NEXT)(1,2,3,4)
@REPEAT(4)(
        DB      @W
        @MATCH(W,NEXT)(@NEXT)
)
```

After expansion:

```
        DB      1
        DB      2
        DB      3
        DB      4
```

In this example ‘,’ is used as a delimiter. Because the number of symbols in (1) is fewer than the substrings, the assignments will be W= “1” and NEXT= “2,3,4.” By using the repeat macro, NEXT can be changed to NEXT= “3,4” then NEXT= “4” and then NEXT=null string.

5.3 Processing Control

5.3.1 Comments

■ **Format** ■

@’text’
    or
@’text *end-of-line*

■ **Function** ■

By using comment macros, comments can be coded anywhere within the source text. A comment macro begins with the metacharacter followed by a single quotation mark (‘). It ends at the next single quotation mark or line feed character (0AH) encountered. When MP encounters a comment macro, it removes all characters from the metacharacter to the final mark.

Even if a macro call is coded within comment text, it will not be handled as a macro call.

There is no limit to the number of characters in comment text.

■ **Example** ■

Before expansion:

```
@DEFINE(MODE)(RUN)           @’ MODE SET
@’************************
@’  MODE SEL
@’************************
@IF(@EQS(@MODE,RUN))THEN(
        MOV     STAT,#1      @’ RUN STATUS’
        CAL     subRUN
)ELSE(
        MOV     STAT,#0      @’ STOP STATUS’
        CAL     subSTOP
)FI
```

After expansion:

```
        MOV     STAT,#1
        CAL     subRUN
```

5.3.2 Brackets

■ **Format** ■

@(*balanced_text*)

■ **Function** ■

The bracket macro removes from macro expansion processing all text from the left parenthesis to the matching right parenthesis. However, escape macros and comment macros will still be effective.

■ **Example** ■

Before expansion:

```
@DEFINE(STYP)(@(@@@@@))
; SPEED @STYP
@SUBSTR(@(1,2,3,4,5),3,5)
```

After expansion:

```
; SPEED @@@@@
2,3,4
```

5.3.3 Escapes

■ **Format** ■

@***n***

■ **Function** ■

The escape macro is called by placing a decimal digit (n=1–9) after a metacharacter. When an escape macro is called, the following *n* characters will not be evaluated. However, escape macros are still effective.

With the escape macro, metacharacters can be inserted as text, commas can be added to text, and a single parenthesis can be placed in a character string that requires balanced parentheses.

■ **Example** ■

Before expansion:

```
@DEFINE(HEADER)(@3))@ DATA @3@(()
;@HEADER

@MACRO(UM1(P1,P2,P3))(
;       P1=<@P1> : P2=<@P2> : P3=<@P3>
)
@UM1(12,34,56,78)
@UM1(12,34@1,56,78)
```

After expansion:

```
;))@ DATA @((

;       P1=<12> : P2=<34> : P3=<56,78>
;       P1=<12> : P2=<34,56> : P3=<78>
```

5.3.4 METACHAR

■ **Format** ■

@**METACHAR**(*balanced_text*)

■ **Function** ■

METACHAR allows the metacharacter to be redefined. The first character of the balanced text will be utilized as the new metacharacter. The balanced text may have any number of characters, but it must not be a null string. The METACHAR call itself expands to a null string.

When the metacharacter is redefined with METACHAR, the character previously defined as the metacharacter will no longer be handled as a metacharacter. The default metacharacter is ‘@.’

■ **Example** ■

The following example defines ‘%’ as the new metacharacter.

```
@METACHAR(%)
```

The following example defines ‘&’ as the new metacharacter.

```
@METACHAR(& NEW META.)
```

5.4 String Comparisons

■ Format ■

@**EQS** (*balanced_text_1*, *balanced_text_2*)
@**NES** (*balanced_text_1*, *balanced_text_2*)
@**LTS** (*balanced_text_1*, *balanced_text_2*)
@**LES** (*balanced_text_1*, *balanced_text_2*)
@**GTS** (*balanced_text_1*, *balanced_text_2*)
@**GES** (*balanced_text_1*, *balanced_text_2*)

■ Function ■

These standard macros compare the two balanced texts delimited by the comma, and depending on the result, they return a string that indicates a logical value. If the result is true, the macro will expand to '-1.' If the result is false, the macro will expand to '00.' Spaces, tabs, and carriage return codes within the texts are also compared.

The comparison of each macro is given below.

EQS	True if both texts are equal.
NES	True if both texts are not equal.
LTS	True if value of first text is less than value of second text.
LES	True if value of first text is less than or equal to value of second text.
GTS	True if value of first text is greater than value of second text.
GES	True if value of first text is greater than or equal to value of second text.

■ Example 1 ■

Before expansion:

```
@LES(ABCD,ABCD)
@LES(abcd,ABCD)
@LES(ABCD,abcd)
```

After expansion:

```
-1
00
-1
```

■ **Example 2** ■

Before expansion:

```
@DEFINE(OPERATION)(MUL)
@IF(@EQS(@OPERATION,ADD))THEN(
        ADD     A,ER0
)ELSE(
  @IF(@EQS(@OPERATION,SUB))THEN(
        SUB     A,ER0
  )ELSE(
        MOV     ER0,A
    @IF(@EQS(@OPERATION,MUL))THEN(
        MUL
    )ELSE(
        DIV
    )FI
  )FI
)FI
```

After expansion:

```
        MOV     ER0,A
        MUL
```

5.5 String Operations

5.5.1 EVAL

■ **Format** ■

@**EVAL**(*expression*)

■ **Function** ■

EVAL returns a string expressing a hexadecimal number. The hexadecimal number is the value obtained by analyzing the expression. The operators described in Section 3.4.2, “Expressions,” can be used within the expression.

The return value is the string expressing the hexadecimal number suffixed with an ‘H.’ If the first character would be an alphabetic letter (A-F), then it will be prefixed with the digit 0 in order to distinguish it from a symbol.

■ **Example** ■

Before expansion:

```
        MOV     A,#@EVAL(1+1)
COUNT1  EQU     @EVAL(33H + 15H + 0F00H)
        ADD     A,#@EVAL(10H-((13+6)*2)+7)
@SET(NUM1,44)
@SET(NUM2,25H)
        AND     A,#@EVAL(@NUM1  <= @NUM2)
        AND     A,#@EVAL(@NUM1 >  @NUM2)
```

After expansion:

```
        MOV     A,#2H
COUNT1  EQU     0F48H
        ADD     A,#-0FH
        AND     A,#0H
        AND     A,#-1H
```

5.5.2 LEN

■ Format ■

@**LEN**(*balanced_text*)

■ Function ■

LEN returns a string expressing a hexadecimal number. The hexadecimal number is the number of bytes that make up the length of the balanced text.

The return value is the string expressing the hexadecimal number suffixed with an 'H.' If the first character would be an alphabetic letter (A-F), then it will be prefixed with the digit 0 in order to distinguish it from a symbol.

■ Example ■

Before expansion:

```
@DEFINE(STR1)(Cross)
@DEFINE(STR2)(Assembler)
        DB      @LEN(123456)
        DB      @LEN(1,2,3)
        DB      @LEN()
        DB      @LEN(@STR1),'@STR1'
        DB      @LEN(@STR2),'@STR2'
        DB      @LEN(@STR1 @STR2),'@STR1 @STR2'
```

After expansion:

```
        DB      6H
        DB      5H
        DB      0H
        DB      5H,'Cross'
        DB      9H,'Assembler'
        DB      0FH,'Cross Assembler'
```

5.5.3 SUBSTR

■ Format ■

@**SUBSTR**(*balanced_text*, *expression_1*, *expression_2*)

■ **Function** ■

SUBSTR returns a specified substring within the text.

Expression 1 specifies the start character position of the substring. Expression 2 specifies the number of characters included in the substring.

If expression 1 is zero or less, or if its value is greater than argument string's length, then SUBSTR will expand to a null string. If expression 2 is zero or less, SUBSTR will expand to a null string. If its value is greater than the remaining length of the string, then SUBSTR will return all characters from the start character to the end of the string.

■ **Example** ■

Before expansion:

```
@DEFINE(FILE)(A:\USR\PROG.SRC)
        DB      '@SUBSTR(ABCDEFG,8,1)'
        DB      '@SUBSTR(ABCDEFG,3,0)'
        DB      '@SUBSTR(ABCDEFG,5,1)'
        DB      '@SUBSTR(ABCDEFG,5,100)'
        DB      '@SUBSTR(123(45)6789,4,4)'
        DB      '@SUBSTR(@FILE,8,@LEN(@FILE)-7)'
```

After expansion:

```
        DB      ''
        DB      ''
        DB      'E'
        DB      'EFG'
        DB      '(45)'
        DB      'PROG.SRC'
```

5.6 Expansion Control

5.6.1 IF

■ **Format** ■

@**IF**(*expression*)**THEN**(*balanced_text_1*)[**ELSE**(*balanced_text_2*)]**FI**

■ **Function** ■

The IF macro evaluates its expression, and expands the text depending on the result.

IF first evaluates the expression. If its value is true (non-zero), then the IF call will expand balanced text 1. If the expression value is false (zero) and an ELSE clause is provided, then the IF call will expand balanced text 2. If the expression value is false but no ELSE clause is provided, then the IF call will return a null string.

An IF call is terminated with FI. IF and FI must balance.

IF calls can be nested. During nesting, an ELSE clause will refer to the newest IF that is still open (not terminated with FI), and an FI will terminate the newest IF that is still open.

■ **Example** ■

Before expansion:

```
@DEFINE(OPERATION)(MUL)
@IF(@EQS(@OPERATION,ADD))THEN(
        ADD     A,ER0
)ELSE(
  @IF(@EQS(@OPERATION,SUB))THEN(
        SUB     A,ER0
  )ELSE(
        MOV     ER0,A
    @IF(@EQS(@OPERATION,MUL))THEN(
        MUL
    )ELSE(
        DIV
    )FI
  )FI
)FI
```

After expansion:

```
        MOV     ER0,A
        MUL
```

5.6.2 IFDEF

■ **Format** ■

@**IFDEF**(*symbol*)**THEN**(*balanced_text_1*)[**ELSE**(*balanced_text_2*)]**FI**

■ **Function** ■

The IFDEF macro evaluates its symbol, and expands the text depending on the result.

If the symbol is currently valid as a user macro or user symbol, then the IFDEF macro will expand balanced text 1. If the symbol is undefined and an ELSE clause is provided, then the IFDEF macro will expand balanced text 2. If the symbol is undefined but no ELSE clause is provided, then the IFDEF macro will return a null string.

■ **Example** ■

Before expansion:

```
@DEFINE(SYM)()
@IFDEF(SYM)THEN(
        ADD     A,ER0
)ELSE(
        MOV     ER0,A
)FI
```

After expansion:

```
        ADD     A,ER0
```

5.6.3 IFUNDEF

■ Format ■

@**IFUNDEF**(*symbol*)**THEN**(*balanced_text_1*)[**ELSE**(*balanced_text_2*)]**FI**

■ Function ■

The IFUNDEF macro evaluates its symbol, and expands the text depending on the result.

If the symbol is not currently valid as a user macro or user symbol, then the IFUNDEF macro will expand balanced text 1. If the symbol is valid and an ELSE clause is provided, then the IFUNDEF macro will expand balanced text 2. If the symbol is valid but no ELSE clause is provided, then the IFUNDEF macro will return a null string.

■ Example ■

Before expansion:

```
@DEFINE(SYM)()
@IFUNDEF(SYM)THEN(
        ADD     A,ER0
)ELSE(
        MOV     ER0,A
)FI
```

After expansion:

```
        MOV     ER0,A
```

5.6.4 WHILE

■ Format ■

@**WHILE** (*expression*) (*balanced_text*)

■ Function ■

The WHILE macro evaluates its expression, and expands the text depending on the result.

WHILE first evaluates the expression. If its value is true (non-zero), then the WHILE call will expand the balanced text. If not, then it will not expand the balanced text. If the balanced text was expanded, then the expression is re-evaluated. If its value is true, then the WHILE call will again expand the balanced text. This operation is repeated until the value of the expression becomes false (zero). Expressions use the operators described in Section 3.4.2, “Expressions.”

Macro processing continues until the expression becomes false, so the balanced text must modify the expression. If it does not, then the WHILE macro cannot terminate.

Expansion can be forcibly terminated with the EXIT macro.

If the balanced text is expanded more than 65535 times, then MP will assume it is in an infinite loop and will forcibly terminate the WHILE macro.

■ **Example** ■

Before expansion:

```
@SET(COUNT,3)
@WHILE(@COUNT > 0)(
        DW      @COUNT
        @SET(COUNT,@COUNT-1)
)
```

After expansion:

```
        DW      3H
        DW      2H
        DW      1H
```

5.6.5 REPEAT

■ **Format** ■

@REPEAT (*expression*) (*balanced_text*)

■ **Function** ■

The REPEAT macro expands the balanced text the specified number of times.

The expression gives the number of times. It is evaluated only one time at the start of the macro call. Expressions use the operators described in Section 3.4.2, “Expressions.”

Expansion can be forcibly terminated with the EXIT macro.

■ **Example** ■

Before expansion:

```
@REPEAT(3)(
        NOP
)
@SET(COUNT,0FFFBH)
@REPEAT(0FFFFH - @COUNT)(
        DW      @COUNT
)
```

After expansion:

```
           NOP
           NOP
           NOP
           DW      0FFFBH
           DW      0FFFBH
           DW      0FFFBH
           DW      0FFFBH
```

5.6.6 EXIT

■ **Format** ■

@EXIT

■ **Function** ■

EXIT forcibly terminates expansion of the most recently called REPEAT, WHILE, or user macro. Generally it is used to prevent infinite loops (for example, a WHILE expression that will never become false or a user macro that recycles endlessly). Multiple EXIT macros can be placed within the same macro.

If an EXIT macro is placed anywhere other than the macro body of a WHILE, REPEAT, or user macro, then MP will recognize it as a syntax error.

■ **Example** ■

Before expansion:

```
   @SET(COUNT,0FFF0H)
   @WHILE(@COUNT < 0FFFFH)(
           @COUNT
           @SET(COUNT,@COUNT+1)
           @IF(@COUNT > 0FFF3H)THEN(
                   @EXIT
           )FI
   )
   @SET(COUNT,0FFF0H)
   @REPEAT(10)(
           @COUNT
           @SET(COUNT,@COUNT+1)
           @IF(@COUNT > 0FFF3H)THEN(
                   @EXIT
           )FI
   )
```

After expansion:

```
0FFF0H
0FFF1H
0FFF2H
0FFF3H
0FFF0H
0FFF1H
0FFF2H
0FFF3H
```

5.7 Console I/O Control

While MP is running, characters can be entered from and output to the console. During output, the color of the characters can be specified.

5.7.1 IN

■ **Format** ■

@IN

■ **Function** ■

The IN macro outputs a '>' to the console as an input prompt, and then returns one line typed at the console (including the carriage return code CR (0DH) and LF (0AH)).

Up to 128 characters can be input. All characters beyond this will be ignored. The carriage return code counts as one character.

■ **Example** ■

This example assumes that 'MODEL-1' is input from the console in response to the IN macro prompt.

Before expansion:

```
;***********************
@OUT(INPUT MODEL-1)
@DEFINE(DATA)(@IN)
; LENGTH = @LEN(@DATA)
; INPUT  = @SUBSTR(@DATA,1,@LEN(@DATA)-1)
;***********************
```

After expansion:

```
;***********************
; LENGTH = 8H
; INPUT  = MODEL-1
;***********************
```

5.7.2 OUT

■ **Format** ■

@**OUT** (*balanced_text*)

■ **Function** ■

The OUT macro outputs the text to the console. The OUT call itself is expanded to a null string.

■ **Example 1** ■

Before expansion:

```
@DEFINE(CR)(
)
@DEFINE(TYPE)(MODEL-3)
@OUT(** COMPILE INFO.
)
@OUT(** TARGET : @TYPE@CR)
```

After expansion (the following will be output to the console):

```
** COMPILE INFO.
** TARGET : MODEL-3
```

■ **Example 2** ■

```
@SET(COUNT,3)
@WHILE(@COUNT>0)(
        DB      @COUNT
        @SET(COUNT,@COUNT)   ...... ①
        @OUT(COUNT = @COUNT) ...... ②
)
```

In this example, at ① @COUNT is written instead of @COUNT-1, which will cause the WHILE macro to enter an infinite loop. By inserting an OUT macro at ②, one can understand the abnormal operation by viewing the screen.

By using the OUT macro to evaluate macro expansion during operation, one can raise development efficiency.

5.7.3 COLOR

■ **Format** ■

@**COLOR** (*balanced_text*)

■ **Function** ■

The COLOR macro specifies the color of characters when output to the console. The first character of the balanced text is used to specify the color. That specified character is a decimal digit corresponding to the following colors.

Specified Character	Color
1	Blue
2	Red
3	Violet
4	Green
5	Cyan
6	Yellow
7	White

The COLOR call itself expands to a null string.

■ **Example** ■

Before expansion:

```
@COLOR(2)
@OUT(
***** Displayed in red *****)
@OUT(@COLOR(5)
***** Displayed in cyan *****)
```

After expansion (the following will be displayed to the console):

```
***** Displayed in red *****
***** Displayed in cyan *****
```

5.8 File Include

While MP is running, contents of other files can be inserted anywhere in the source text. The inserted contents are handled as if they had been at that location from the beginning.

There is no limit to the number of included files. However, when other files are included within included files, the depth of inclusion nesting is limited to 13 levels.

5.8.1 INCLUDE

■ **Format** ■

@**INCLUDE**(*file_specification*)

■ **Function** ■

INCLUDE inserts the contents of the specified file at the current location.

If a path is specified in the file specification, then MP will search for the file in that path. If no path is specified, then MP will search the current directory.

In addition, if a path is specified with the IC option when MP is invoked, then this path will be used. If a path is then specified in the file specification, then it will be ignored. For example, assume the following text.

```
@INCLUDE(B:\USER\SUB.ASM)
```

Also assume that MP is invoked with the following.

```
MP MAIN.ASM IC(C:\INC\)
```

Then despite the text specification, the referenced path will be C:\INC\ instead of B:\USER\.

5.8.2 MACROLIB

■ **Format** ■

@**MACROLIB**(*file specification*)

■ **Function** ■

MACROLIB extracts the macro definitions of the specified file and incorporates them in MP's symbol table. Unlike INCLUDE, MACROLIB disregards all other parts of the file (expansions and other strings). By using MACROLIB, one can execute a series of user macro definitions in with one macro, and easily manage them as a group.

The MACROLIB macro itself expands to a null string.

If a path is specified in the file specification, then MP will search for the file in that path. If no path is specified, then MP will search the current directory.

In addition, if a path is specified with the ML option when MP is invoked, then this path will be used. If a path is then specified in the file specification, then it will be ignored. For example, assume the following text.

```
@MACROLIB(B:\USER\SUB.ASM)
```

Also assume that MP is invoked with the following.

```
MP MAIN.ASM ML(C:\INC\)
```

Then despite the text specification, the referenced path will be C:\INC\ instead of B:\USER\.

INCLUDE and MACROLIB macros must be directly coded at the locations where their expansion results will be in the output file. In other words, file contents incorporated with INCLUDE or MACROLIB cannot be passed directly to symbols, such as parameters.

For example, the following example is valid.

```
@IF(1)THEN(
       @INCLUDE(LIB.H)
)FI
```

However, the example below assigns the expansion results to SYM, so it is invalid.

```
@DEFINE(SYM)(@INCLUDE(LIB.H))
```

5.9 Listing Control

5.9.1 GEN, GENONLY

■ **Format** ■

@**GEN**
@**GENONLY**

■ **Function** ■

These macros specify the output format of expanded text.

GENONLY specifies that only expanded text is to be output.

GEN specifies that both expanded text and all macro definitions are to be output. The mark specified with the GEN option when MP was invoked will be inserted at the start of the definition line. The default mark is ";++."

These specifications are effective until the next GEN or GENONLY macro is called within the text.

GEN and GENONLY themselves expand to null strings.

■ **Example** ■

Before expansion:

```
@GEN
@MACRO(GETP(DX1,DX2))(
        MOVB    R0,#@DX1
        MOVB    R1,#@DX2
)
        MOV     A,#50H
@GETP(10,20)

        RT
;------------------------
@GENONLY
@MACRO(GETP(DX1,DX2))(
        MOVB    R0,#@DX1
        MOVB    R1,#@DX2
)
        MOV     A,#50H
@GETP(30H,40H)

        RT
```

After expansion:

```
;++@MACRO(GETP(DX1,DX2))(
;++     MOVB    R0,#@DX1
;++     MOVB    R1,#@DX2
;++)
        MOV     A,#50H
;++@GETP(10H,20H)
        MOVB    R0,#10H
        MOVB    R1,#20H

        RT
;----------------------------
;++@GENONLY
        MOV     A,#50H
        MOVB    R0,#30H
        MOVB    R1,#40H

        RT
```

5.10 System Control

5.10.1 SYSTEM

■ **Format** ■

@**SYSTEM** (*balanced text*)

■ **Function** ■

SYSTEM executes the command specified by the text (including internal commands).

The SYSTEM call itself expands to a null string.

■ **Example** ■

Before expansion:

```
@SYSTEM(DIR/W)
```

After expansion:

The DIR command is executed, and the directory is displayed on the console.

5.10.2 EXIST

■ **Format** ■

@**EXIST** (*file specification*)

■ **Function** ■

EXIST determines whether or not the specified file exists, and returns a logical value depending on the result. If the file exists, EXIST will expand to true (‘-1’). If it does not exist, EXIST will expand to false (‘00’).

■ **Example** ■

Before expansion:

```
@DEFINE(CR)(
)
@OUT(@CR***** INCLUDE FILE SELECT *****@CR)
@WHILE(1)(
        @COLOR(7)
        @OUT(Enter File Name:)
        @DEFINE(WORK)(@IN)
        @DEFINE(NAME)(@SUBSTR(@WORK,1,@LEN(@WORK)-1))
        @IF(@EXIST(@NAME))THEN(
                @COLOR(5)
                @OUT(”@NAME” FILE EXIST.@CR)
                @EXIT
        )
        ELSE(
                @COLOR(2)
                @OUT(”@NAME” FILE NOT FOUND.@CR)
        )FI
)
@INCLUDE(@NAME)
```

After expansion:

The file specified by console input will be included.

5.10.3 SOURCE

■ **Format** ■

@SOURCE

■ **Function** ■

SOURCE expands to the name of the source file that was specified when MP was invoked.

■ **Example** ■

Command line invoking MP:

```
MP PRG.SRC
```

Before expansion:

```
        DB    ’@SOURCE’
```

After expansion:

```
        DB    ’PRG.SRC’
```

5.10.4 FDATE

■ **Format** ■

@FDATE

■ **Function** ■

FDATE expands to an 8-character string that indicates the date of the source file that was specified when MP was invoked.

Only the lower two digits of the year are returned. The date display format is as follows.

xx-xx-xx (year-month-date)

■ **Example** ■

Before expansion:

```
          DB    '@FDATE'
```

ter expansion:

```
          DB    '87-10-21'
```

5.10.5 FTIME

■ **Format** ■

@FTIME

■ **Function** ■

FTIME expands to an 5-character string that indicates the time of the source file that was specified when MP was invoked. The time display format is as follows.

xx:xx (hour:minute)

■ **Example** ■

Before expansion:

```
   @FTIME
```

After expansion:

```
   12:16
```

5.10.6 DATE

■ Format ■

@DATE

■ Function ■

DATE expands to an 8-character string that indicates the current date of the operating system.

Only the lower two digits of the year are returned. The date display format is as follows.

xx-xx-xx (year-month-date)

■ Example ■

Before expansion:

```
        DB      '@DATE'
```

After expansion:

```
        DB      '87-10-21'
```

5.10.7 TIME

■ Format ■

@TIME

■ Function ■

TIME expands to an 5-character string that indicates the current time of the operating system. The time display format is as follows.

xx:xx (hour:minute)

■ Example ■

Before expansion:

```
@TIME
```

After expansion:

```
12:16
```

5.11 Purging Symbol Definitions

5.11.1 PURGE

■ **Format** ■

@**PURGE** (*symbol* [, *symbol*...])

■ **Function** ■

PURGE purges the record of the specified symbols. The symbols must be currently defined user symbols or user macro names.

When two or more symbols are specified, delimit them with commas (,).

The PURGE call itself expands to a null string.

■ **Example** ■

Before expansion:

```
@DEFINE(SYM)(TEXT)
@IFDEF(SYM)THEN(
        SYM IS DEFINED
)ELSE(
        SYM IS UNDEFINED
)FI
;
@PURGE(SYM)
@IFDEF(SYM)THEN(
        SYM IS DEFINED
)ELSE(
        SYM IS UNDEFINED
)FI
```

After expansion:

```
        SYM IS DEFINED
;
        SYM IS UNDEFINED
```

5.11.2 ALLPURGE

■ **Format** ■

@**ALLPURGE**

■ **Function** ■

ALLPURGE purges all currently defined user symbols or user macro names. The ALLPURGE call itself expands to a null string.

■ Example ■

Before expansion:

```
@DEFINE(SYM1)(TEXT1)
@DEFINE(SYM2)(TEXT2)
@DEFINE(SYM3)(TEXT3)
@DEFINE(SYM4)(TEXT4)
@ALLPURGE
;
@IFUNDEF(SYM1)THEN(SYM1 IS UNDEFINED)FI
@IFUNDEF(SYM2)THEN(SYM2 IS UNDEFINED)FI
@IFUNDEF(SYM3)THEN(SYM3 IS UNDEFINED)FI
@IFUNDEF(SYM4)THEN(SYM4 IS UNDEFINED)FI
```

After expansion:

```
;
SYM1 IS UNDEFINED
SYM2 IS UNDEFINED
SYM3 IS UNDEFINED
SYM4 IS UNDEFINED
```

6. Error Messages

Errors fall into two categories, macro errors and fatal errors, depending on their characteristics.

Macro errors allow processing to continue. Mainly syntax errors are classified as macro errors. When a macro error occurs, MP displays a message on the console and continues processing.

Fatal errors do not allow processing to continue. Mainly I/O errors are classified as fatal errors. When a fatal error occurs, MP displays a message on the console and immediately suspends processing.

6.1 Macro Error Messages

Macro error messages are displayed in the following format.

file_name (*line_number*) : error *error_number* : *message*

If there is an error on the command line, then the following message will be output.

command error *error_number* : *message*

The meaning of each item is given below.

file_name	Name of the source file that generated the error.
line_number	Line number in the source file that generated the error.
error_number	A number indicating the type of error.
message	Brief message describing the error.

The following list of messages is given in order of error numbers.

name	Name of the macro that generated the error.
character	Character that caused the error.

No. **Message**

00 **undefined macro name :** *name*

An undefined macro name was called, or a PURGE macro specified an undefined symbol.

01 **bad macro specification**

The characters following a metacharacter were not recognized as a macro name.

02 **bad macro position :** *name*

A command option was used within the source text, or a standard macro was used as a command option.

03 **missing balanced text**

A problem exists with the balanced text of a standard macro. Text may be missing, or parentheses within the text may not balance.

04 **missing "THEN" in "IF"**

The keyword THEN has been left out of an IF, IFDEF, or IFUNDEF macro.

05 **missing "FI" in "IF"**

The keyword FI has been left out of an IF, IFDEF, or IFUNDEF macro.

06 **too long expanded line**

The number of characters in one line of output text exceeds 4096 characters. All further characters beyond 4096 will be ignored.

07 **bad symbol or symbol list format**

A macro name, parameter list, or local list coded in a MACRO macro, or a symbol coded in a MATCH, SET, DEFINE, IFDEF, IFUNDEF, or PURGE macro contains an error.

08 **bad COLOR macro character :** *character*

The character specifying color for the COLOR macro is not '1'–'7.'

09 **bracket macro not closed**

No right bracket was found for a bracket macro.

10 **bad filename or path name**

The file specification or path specification for an EXT, ERRORPRINT, SPA, SPB, INCLUDE, or MACROLIB option was incorrect, or the file specification for an INCLUDE, MACROLIB, or EXIST macro was incorrect.

11 **ERRORPRINT already specified**

The ERRORPRINT option has already been specified.

12 **EXT already specified**

The EXT option has already been specified.

13 **SPA/SPB already specified**

The SPA or SPB option has already been specified.

14 **INCLUDE or MACROLIB already specified**

The INCLUDE or MACROLIB option has already been specified.

15 **INCLUDE/MACROLIB bad position**

An INCLUDE or MACROLIB macro is coded at an incorrect location.

16 **INCLUDE/MACROLIB nesting too deep**

Nesting for INCLUDE or MACROLIB macros exceeds 13 levels.

17 **illegal attempt to define macro :** *name*

A standard macro name would be used as a user macro name, user symbol, parameter, or local label.

18 **redefined parameter or label in this macro :** *name*

A parameter or local label with the same name would be used in the same user macro.

19 **illegal expression**

An expression is coded incorrectly.

20 **divided by zero**

A division by 0 is coded within an expression.

21 **value overflow**

The calculated result of an expression exceeds 0FFFFH.

22 **illegal EXIT macro**

An EXIT macro was not coded in a WHILE, REPEAT, or user macro.

23 **missing delimiter :***character*

A delimiter character could not be found in a user macro call.

24 **illegal meta_character :** *character*

An inappropriate character was specified as the metacharacter with the METACHAR macro.

25 **non stop loop in WHILE**

The number of loops of a WHILE macro exceeded 0FFFF times.

6.2 Fatal Error Messages

Fatal error messages are displayed in the following format.

fatal error *error_number* : *message*

The meaning of each item is given below.

error_number	A number indicating the type of error.
message	Brief message describing the error.

The following list of messages is given in order of error numbers.

filename	Name of the file that generated the error.

No.	Message
01	**file not found :** *filename* The source file or include file could not be found.
02	**can not open file :** *filename* The expanded file, symbol file, or error file could not be opened.
03	**not enough memory** There is not enough memory to continue processing.
04	**file seek error** An error occurred during a file seek.
05	**file close error :** *filename* The file could not be closed. Disk capacity could be insufficient.
06	**internal error** A hardware problem or an MP internal error occurred.

Appendix

A. Standard Macro Table

@MACRO(*macro_name* [*parameter_list*])[*local_list*](*macro body*)
@DEFINE(*symbol*)(*balanced_text*)
@SET(*symbol, expression*)
@MATCH(*symbol*[,*symbol...*])(*balanced_text*)
@'text' or *end-of-line*
@(*balanced_text*)
@n
@METACHAR(*balanced_text*)
@EQS(*balanced_text_1*, *balanced_text_2*)
@NES(*balanced_text_1*, *balanced_text_2*)
@GTS(*balanced_text_1*, *balanced_text_2*)
@GES(*balanced_text_1*, *balanced_text_2*)
@LTS(*balanced_text_1*, *balanced_text_2*)
@LES(*balanced_text_1*, *balanced_text_2*)
@EVAL(*expression*)
@LEN(*balanced_text*)
@SUBSTR(*balanced_text*, *expression_1*, *expression_2*)
@IF(*expression*)THEN(*balanced_text*)[ELSE(*balanced_text*)]FI
@IFDEF(*symbol*)THEN(*balanced_text*)[ELSE(*balanced_text*)]FI
@IFUNDEF(*symbol*)THEN(*balanced_text*)[ELSE(*balanced_text*)]FI
@WHILE(*expression*)(*balanced_text*)
@REPEAT(*expression*)(*balanced_text*)
@EXIT
@IN
@OUT(*balanced_text*)
@COLOR(*balanced_text*)
@INCLUDE(*file_specification*)
@MACROLIB(*file_specification*)
@GEN
@GENONLY
@SYSTEM(*balanced_text*)
@EXIST(*file_specification*)
@SOURCE
@FDATE
@FTIME
@DATE
@TIME
@PURGE[*symbol*[,*symbol...*]]
@ALLPURGE

B. Option Table

The table below shows the options that can be used on the invoking command line.

Option Format	Abbreviated Form	Function
ERRORPRINT[(file_specification)]	EP	Specifies output destination of error messages.
EXT[(file_specification)]	—	Specifies output destination of text after macro expansion.
SPA[(file_specification)]	—	Specifies output of symbol table contents.
SPB[(file_specification)]	—	Specifies output of symbol table contents.
INCLUDE(path_specification)	IC	Specifies path name of include files.
MACROLIB(path_specification)	ML	Specifies path name of include files.
GEN[(mark_specification)]	GE	Specifies the format of output text.
GENONLY	GO	Specifies only expanded lines as output text.
DEFINE(symbol)(string)	DEF	Defines a user symbol.
DL	—	Eliminates white space created by macro expansion.

Note: '—' indicates that no abbreviated form exists.

The following options are specified as defaults when MP is invoked.

Option	Function
ERRORPRINT	Output to console.
EXT	Output to a file with the source file name and extension .Q.
GENONLY	Output expanded lines only.

C. Function Changes From Ver. 1.0X

1. Invoke time options

- The DL option has been added. White space created by macro expansion is eliminated when the DL option is specified.
- The source line mark when GEN is specified has been changed. Also, the mark can be specified when MP is invoked.
- The file specification may be omitted with the EXT and ERRORPRINT options.

2. Standard macros

- The PURGE and ALLPURGE macros have been added.
- The bracket macro has been added.
- The IFDEF and IFUNDEF macros have been added.
- The number of user macro parameters and local labels is unlimited.
- Any character can be used as the user macro parameter delimiter.
- The INCLUDE and MACROLIB macros are not affected by the system configuration.
- The final character (carriage return code) of the IN macro counts as one character.
- The number of characters output by the OUT macro is unlimited.
- The coding location of all macros except EXIT, INCLUDE, and MACROLIB is unrestricted.

3. General coding

- The number of characters in standard macro text or user macro real parameters is unlimited.
- Multiple expansion of macros (@@macro_name) is now possible.

4. Output files and screen output

- The symbol file format has been updated.
- The error message output format has been updated.
- Macro error line numbers display the number of the first line of the macro that generated the error.
- In Version 1.00 a buzzer sounded when MP terminated if there had been format errors, and the termination message was displayed in red. These functions have been removed.