USER'S MANUAL

NEC

RA78K SERIES

STRUCTURED ASSEMBLER PREPROCESSOR

PC-9800 SERIES (MS-DOS™ BASED)

IBM (PC-DOS™ BASED)

Document No. EEU-1254A
(O.D.No. EEU-643C)
Date Published September 1990P
Printed in Japan

USER'S MANUAL

NEC

RA78K SERIES

STRUCTURED ASSEMBLER PREPROCESSOR

PC-9800 SERIES (MS-DOS BASED)

IBM (PC-DOS BASED)

© NEC Corporation 1990

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Corporation. NEC Corporation assumes no responsibility for any errors which may appear in this document.

NEC Corporation does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from use of a device described herein or any other liability arising from use of such device. No license, either express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Corporation or of others.

INTRODUCTION

This manual is designed to facilitate correct understanding of the functions, the method of describing a source program, and the operating procedures of the structured assembler preprocessor (hereinafter referred to as the "structured assembler") included in the 78K Series Microcomputer Assembler Package "RA78K Series". This manual does not explain the other programs included in the assembler package. Therefore, when you write a program using this structured assembler, also read the 78K series Assembler Package User's Manual (both for Language and Operation) for the applicable microcomputer.

[Target Devices]
All devices to which the RA78K series assembler package is applicable

[Readers of Manual]
This manual is intended for those who are familiar with the functions of the 78K series microcomputers. For the functions of the 78K series microcomputers, refer to the user's manual published for each device.

[Organization of Manual]

This manual consists of the following 10 chapters:

Chapter 1 - General

Outlines the role and functions of the structured assembler in the development of software for the microcomputer.

Chapter 2 - How to Describe Source Program

Describes the general rules applicable to the description of a source program such as the basic configuration and description format of source programs, reserved words, etc.

Chapter 3 - Control Statements

Describes the functions, description formats, and application examples of the respective control statements which represent the structure of a program such as if-else-endif and of the respective condition expressions which specify the condition of each control statement.

Chapter 4 - Expression Statements

Describes the functions, description formats, and application examples of the respective expression statements for such operations as substitution, decrement/increment, and exchange.

Chapter 5 - Directives

Describes the functions, description formats, and application examples of pseudo-instructions such as #define and #ifdef which function similarly to macros and are referred to as "directives".

Chapter 6 - Product Overview

Outlines the filenames and operating environment of the structured assembler.

Chapter 7 - Operating Procedures

Details the option functions of the structured assembler and how to start up the structured assembler.

Chapter 8 - Input/Output Files

Outlines the input files (input source module file and include file) and output files (secondary source module file and error list file) of the structured assembler.

Chapter 9 - Error Messages and Termination Information

Describes the respective error messages to be output by the structured assembler and return codes to be output on termination of processing by the structured assembler.

Chapter 10 - List of Limit Values

Contains the various limit values to be noted when using the structure assembler.

Appendixes

Contains a list of statement structures, a list of operands, and a list of instructions to be generated.

[Recommended Usage of Manual]

Chapters 2 through 5 of this manual are related to programming with the structured assembler and should be used when you develop a program.

Chapters 6 through 10 of the manual are related to the operation of the structured assembler and should be used when executing the structured assembler.

For those who use the structured assembler for the first time, read the manual from Chapter 1, General.

For those who have a general understanding of the structured assembler, Chapter 1 may be skipped. However, it is advisable to read Section 1.3 , "Reminders Before Program Development".

[Symbols and Abbreviations]

The following symbols and abbreviations are used in this manual:

Symbol	Meaning
...	: Continuation (repetition) of data in the same format
[ ]	: Parameter(s) in brackets may be omitted.
" "	: A character or character string enclosed in " " (double quotes) must be input as is.
' '	: A character or character string enclosed in ' ' (single quotes) must be input as is.
_____	: The underlined part (character string) must be input by the user from the keyboard.
:	: This part of the program description is omitted.
( )	: A character or character string enclosed in parentheses must be input as is.
CR	: Carriage Return
LF	: Line Feed
/	: Delimiter
α	: Parameter (e.g., a register name) to be described as the operand of a mnemonic instruction
β	: Parameter (e.g., a register name) to be described as the operand of a mnemonic instruction
γ	: Parameter (e.g., a register name) to be described as the operand of a mnemonic instruction
δ	: Parameter (e.g., a register name) to be described as the operand of a mnemonic instruction
$\triangle$	: One or more Blank (Space) characters

TABLE OF CONTENTS

APPENDIXES

CHAPTER 1. GENERAL

1.1 Overview of Structured Assembler

The 78K series structured assembler preprocessor is a program which is used for the development of software for the 78K series microcomputers and is included in the RA78K series assembler package.

In this manual, any of the following 78K series structured assembler preprocessors is referred to as the "structured assembler" or "ST78K" (short for STructured assembler).

ST78K0 .... Structured assembler for RA78K0
ST78K1 .... Structured assembler for RA78K1
ST78K2 .... Structured assembler for RA78K2
ST78K3 .... Structured assembler for RA78K3
ST78K6 .... Structured assembler for RA78K6

The structured assembler converts "if-else-endif", "for-next" and other blocks which represent the structure of a program, into the assembler's source program. Blocks "if-else-endif", "for-next", etc. are described by using control statements.

The structured assembler has the following three advantages:

① Programs can be written easily.
- o A program can be written without changing its structure and thus easy coding is assured from the design stage.
- o No label name is required for branching.
- o A conventional move instruction requiring lengthy descriptions can be described in the form of a substitution statement.

② Programs can be read easily.
- o The structure of a program can be made clear.
- o An instruction to execute an arithmetic operation or data transfer between a memory and a register can be written with a single statement.
- o Programs written by other programmers can be read more easily.
- o Program maintenance or modifications can be accomplished with a greater ease.

③ Desk debugging can be performed easily.

o Because the structured assembly language allows statement description in one-to-one correspondence to the detailed design, desk debugging can be performed with minimal effort.

1.2 Functional Outline

The structured assembler analyzes various control statements, expressions, and directives in a source program written based on the language specifications of the structured assembly language, and outputs an assembler source program which becomes an input source module file to the assembler.

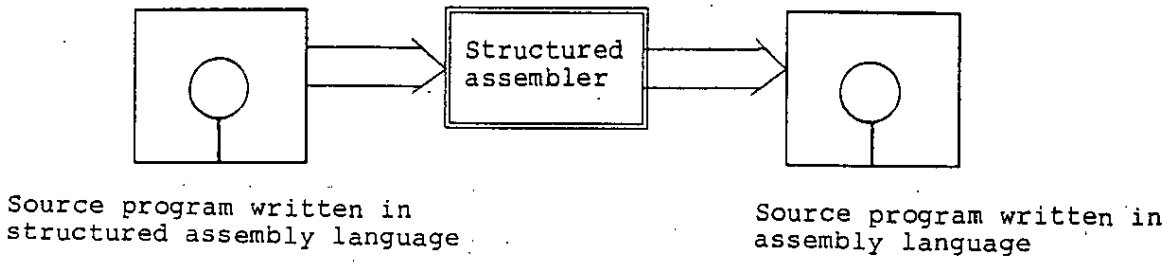


Fig. 1-1. Flow of Structured Assembler

To a secondary source module file, structured statements as comment statements, assembler instructions after the conversion, and ordinary assembly language are output.
If any error exists, an error message is also output.

The main features of the structured assembler preprocessor are as follows:

o Programs can be described easily thanks to abundant control structures in C language style.
o Substitution statements, substitution operators, etc. all in C language style can be described.
o Control structures and substitution statements can be used for bit processing.
o C language styled directives include those for identifier definition, conditional processing, and Include file specification.

- Because the preprocessor outputs assembler source programs, optimization of code can be executed after the conversion by the structured assembler.
- Because a directive is provided to converted a CALL instruction into a CALLT instruction, routines which are to be stored in the CALLT table after the program development can be determined.
- An easy-to-see assembly list can be generated by changing the output position of the assembler source program.

Fig. 1-2 shows the flow of a program development process with the structured assembler.

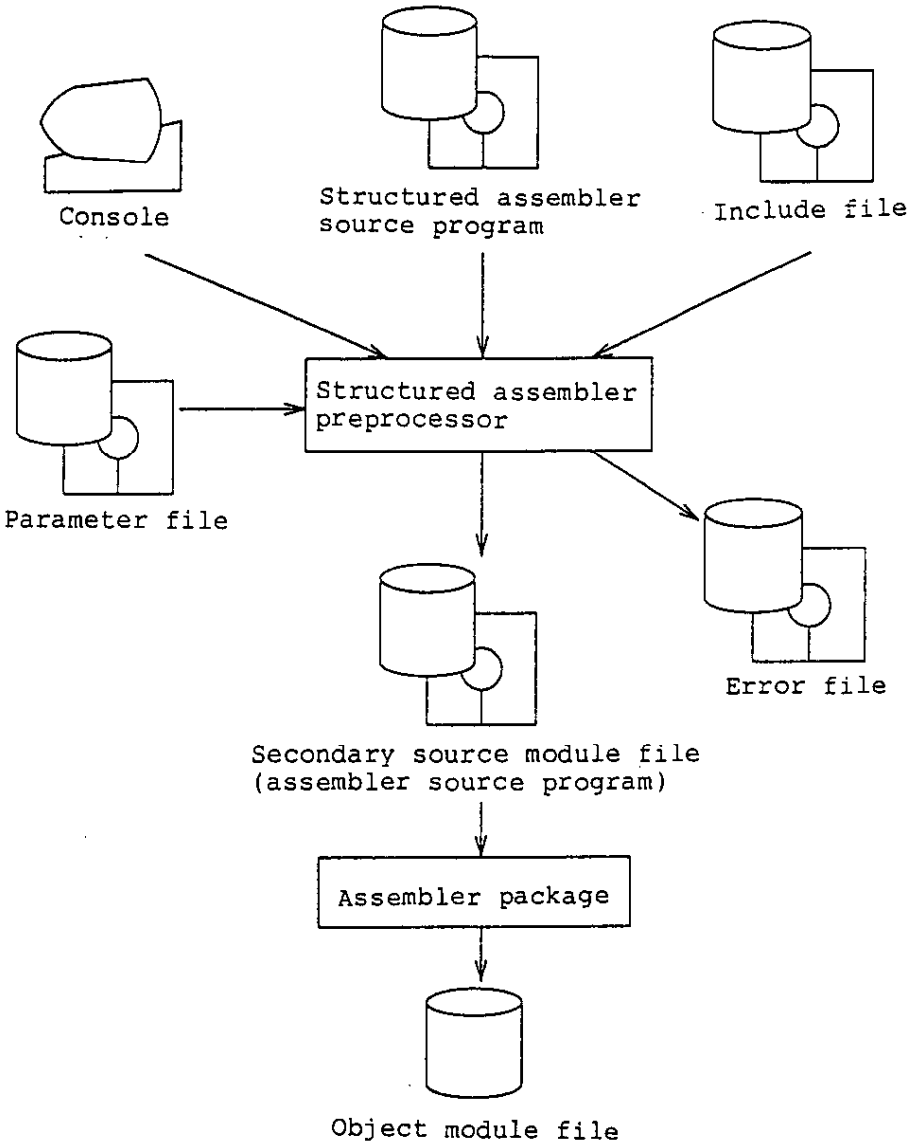


Fig. 1-2. Flow of Program Development Process

1.3 Reminders Before Program Development

The limit values of the structured assembler are as listed in table in 1.3.1 below. Before you start writing a program, keep your mind on these limit values as well as the hints on use of the structured assembler described in 1.3.2 below.

1.3.1 Limit Values

The structured assembler has the following limit values:

Item	Restriction (Max. value)
No. of characters per line (excluding LF or CR)	120 characters
No. of symbols (identifiers) that can be registered with **#define** directive (excluding reserved words)	512 symbols
Nesting level of control statements	31 levels
Nesting level of Conditional processing (**#ifdef**) directives	8 levels
Number of CALLT substituting (**#defcallt**) directives that can be described per source program	32 times
Nesting level of Include (**#include**) directives	No nesting is allowed.

1.3.2 Hints on use

(1) Instructions to be generated and application examples

The instructions to be generated and application examples described in this manual are those applicable to the ST78K2 structured assembler. For other structured assemblers in the 78K series, see Appendix C, "List of Generated Instructions".

(2) Word symbol and byte symbol

The structured assembler bases the last character of a symbol name to make judgment on whether the symbol is a word symbol or byte symbol. This last character which indicates a word symbol can be changed by the SC option. If this option is not specified, "P" or "p" is assumed as the last character. See 7.4, Structured Assembler Options for the SC option.

Example 1: Structured assembler

```
SYM = #3
SYMP= #3
```

Assembler

```
MOV   SYM,#3
MOVW  SYMP,#3
```

Example 2: Start-up command of structured assembler

```
A>ST78K3 INPUT.A -SC@
         |       |
         |       Specifies @ as a character
         |       which indicates a word symbol.
         Input file specification
Command name of structured assembler
```

Structured assembler

```
SYMP = #3
SYM@ = #3
```

Assembler

```
MOV   SYMP,#3
MOVW  SYM@,#3
```

(3) Label definition

If a label (i.e., a symbol indicating an address in the assembler) is to be defined, describe the label definition separate from the statement of the structured assembler.

Example: Acceptable

```
SYMBOL:
        AX=#10H
```

Not acceptable

```
SYMBOL:  AX=#10H
```

CHAPTER 2. HOW TO DESCRIBE SOURCE PROGRAMS

2.1 Basic Configuration of Source Program

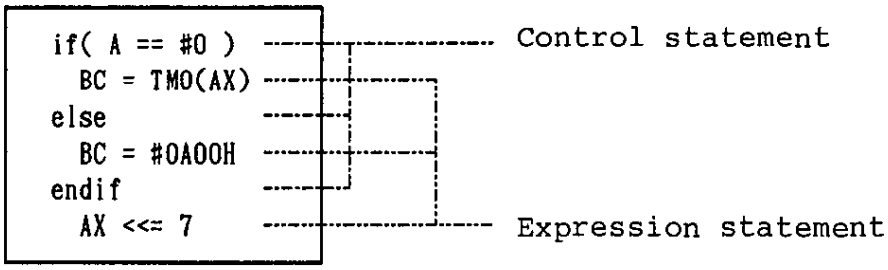
A source program (or source module) is written in both the structured assembly language and the assembly language.
See the RA78K series Assembler Package User's Manual for the assembly language.
Up to 120 characters may be described per source program line (between one LF character and the next).
The structured assembly language has the following types of descriptions shown in Table 2-1 below.

Table 2-1. Descriptions of Structured Assembly Language

Type of statement			Description
Structured assembler statements	Control statements	Conditional branch	if-elseif-else-endif if_bit-elseif_bit-else-endif switch-case-default-ends
		Condition loop	for-next while-endw while_bit-endw repeat-until repeat-until_bit
		Others	break, continue, goto
	Expression statements	Substitution	=, +=, -=,*=, /=, &=, =, >>=, <<=
		Counter	++, --
		Exchange	<->
Condition expressions	Relational operators		==, !=, <, >, >=, <=
	Bit conditions		bit address, !bit address
	Logical operators		AND (&&), OR (||)

Each condition expression is used as the condition of a control statement. (See Section 3.4, "Functions of Control Statements" in Chapter 3 for details.)

Example:



(1) Control statements

Control statements include **if** and **switch-case** statements which specify conditional branching, **for-next**, **while**, and **repeat-until** statements which specify conditional looping, and **break**, **continue**, and **goto** statements which specify exit from a loop. For details, see Chapter 3, Control Statements.

(2) Expression statements

Expression statements include substitution statements, counter (increment/decrement) statements, and exchange statement. For details, see Chapter 4, Expression Statements.

2.2 Constituent Elements of Source Program

(1) Character set

The following alphabetic, numeric, and special characters can be used to describe a source program.

Alphabetic characters

```
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
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
```

Numeric characters

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

Special characters

Character	Name	Main use
	Blank	Delimiter of each field (character or character string)
?	Question mark	Symbol equivalent to alphabetic characters
@	Unit price symbol	Symbol indicating the start of indirect addressing
_	Underscore	Symbol equivalent to alphabetic characters
,	Comma	Delimiter between operands
.	Period	Bit position specifier of bit symbol
+	Plus sign	Positive sign or ADD operator
-	Minus sign	Negative sign or SUBTRACT operator
*	Asterisk	MULTIPLY operator
/	Slash	DIVIDE operator
&	Ampersand	AND operator
|	Separator	OR operator
^	Caret	XOR operator
( )	Left and right parentheses	Symbol pair indicating the order of precedence of operations to be performed or indicating a control statement

Character	Name	Main use
=	Equal sign	SUBSTITUTE operator or Relational operator
;	Semicolon	Symbol indicating the start of a Comment field or delimiter between expressions in a control statement
:	Colon	Delimiter between labels
#	Sharp sign	First character of each structured assembler directive or symbol indicating Immediate addressing mode
!	Exclamation mark	Symbol indicating the start of Direct addressing or symbol indicating negation
< >	Not equal sign	Relational operators (Less-Than and Greater-Than)
¥	Yen sign	Symbol indicating a directory (with MS-DOS)
\	Back slash	Symbol indicating a directory (with PC-DOS)
$	Dollar sign	Symbol indicating the location counter value or symbol indicating a control instruction
HT	Horizontal tab	Character equivalent to Blank
LF	Line feed	Symbol indicating the end of a line

(2) Identifiers

Identifiers refer to names given to numeric data or addresses. By using these identifiers in a source program, the user can read the contents of the source program more easily. Each identifier must be defined with a **#define** directive. (See Section 5.2, "Function of Each Directive" in Chapter 5 for details.)

(3) Symbols

The structured assembler bases the last character of a symbol name to make judgment on whether a byte-access or word-access instruction is to be generated for the symbol. This last character specification for a symbol name may be changed with the SC option. If this option is omitted, "P" or "p" (meaning Pair) is assumed as the word symbol.

The structured assembler has no symbol. Thus, symbols written in the assembly language are output as character strings and without change to a secondary source module file. (For the symbols of the assembly language, see the RA78K Series Assembler Package User's Manual.)

To define any of the labels written in the assembly language, the label must be described separate from the statement of the structured assembler.

(4) Constants

The structured assembler has no constant. Thus, constants written in the assembly language are output as character strings and without change to a secondary source module file. (For the constants of the assembly language, see the RA78K Series Assembler Package User's Manual.)

(5) Expressions

Each expression consists of constant(s), special character(s), and/or symbol(s) connected with an operator. (For the expressions of the assembly language, see the RA78K Series Assembler Package User's Manual.)

If an expression written in the assembly language uses Blank as a delimiter between operands, enclose the expression in a pair of parentheses when describing it on the source program of the structured assembler.

Example:

① To describe expressions on Assembler

```
MOV     A, #SYM AND 0FFH
MOV     A, LABEL + 1
```

② To describe expressions on source program of Structured Assembler

```
A = #(SYM AND 0FFH)
A = (LABEL + 1)
```

2.3 Reserved Words

The reserved words of the structured assembly language are as listed in the table below.

Table 2-2. List of Reserved Words

Control statements	if switch for while repeat forever break		if_bit case next while_bit until continue		elseif default endw until_bit goto		elseif_bit ends		endif
Directives	#define #ifdef #include #defcallt		#else #endcallt		#endif		define ifdef include defcallt	else endcallt	endif
Operators	++ *= == +	-- /= != -	+= <<= < *	-= >>= >= /	&= > &	|= <= |	^= ^		
Flags/ registers/ SFR	78K/0	1 bit	CY	Z					
		8 bits	X R0	A R1	C R2	B R3	E R4	D R5	L R6	H R7
		16 bits	AX CR00 PR0	BC CR01 PR1	DE TM0	HL SP	RP0 IF0	RP1 IF1	RP2 MK0	RP3 MK1
	78K/I	1 bit	CY	Z						
		8 bits	X R0	A R1	C R2	B R3	E R4	D R5	L R6	H R7
		16 bits	AX RP2 CR00 CPT0 PWM0 CPT2H	BC RP3 CR01 CPT1 PWM1	DE CR02 CPT2 FRC	HL CR10 CPT3 IF0	SP CR11 TM0 MK0	RP0 CR12 TM1 PR0	RP1 CR20 TM2 ISM0	
									(See Note below.)	
	78K/II	1 bit	CY	Z						
		8 bits	X R0	A R1	C R2	B R3	E R4	D R5	L R6	H R7
		16 bits	AX RP3 CR00 ISM0	BC CR01	DE CR02	HL TM0	RP0 IF0	RP1 MK0	RP2 PR0	

NOTE: With the uPD78112, SP becomes an 8-bit register.

Flags/ registers/ SFR (contd)	78K/III	1 bit	CY Z						
		8 bits	X A C B E D L H R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 VPL VPH UPL UPH						
		16 bits	AX BC DE HL SP VP UP RP0 RP1 RP2 RP3 RP4 RP5 RP6 RP7 ADCR ADCR0 ADCR1 ADCR2 ADCR3 ADCR4 ADCR5 ADCR6 ADCR7 BRG CC00R CC01LW CC01R CC01UW CC10 CCX0LW CCX0UW CM00 CM00R CM00S CM01 CM01R CM01S CM02 CM02R CM02S CM03 CM03R CM03S CM04R CM04S CM05R CM05S CM06S CM10 CM11 CM12 CM20 CM21 CM30 CMX0 CPT0 CPT1 CR00 CR01 CR10 CR11 CSE0 CSE1 CT00 CT01 CT01LW CT01UW CT02 CT02LW CT02UW CT03LW CT03UW CTX0LW CTX0UW IF0 IF1 ISM0 ISM1 MD0 MD1 MK0 MK1 PB0 PB1 PWM0 PWM1 PWMB PWMC TM0 TM0LW TM0UW TM1 TM2 TM3 UDC0 UDC1						
	78K/VI	1 bit	CY Z						
		8 bits	R0L R0H R1L R1H R2L R2H R3L R3H R4L R4H R5L R5H R6L R6H R7L R7H						
		16 bits 32 bits	R0 R1 R2 R3 R4 R5 R6 R7 RP0 RP1 RP2 RP3 ADCR0 ADCR1 ADCR2 ADCR3 ADCR4 ADCR5 ADCR6 ADCR7 CC00LW CC01LW CC02LW CC03LW CC00UW CC01UW CC02UW CC03UW CC10 CC11 CM00 CM01 CM02 CM03 CM10 CM11 CM20 CM21 CSE0 CT20 CT21 IF0 IPGCM0 IPGCM1 IPGCM2 IPGCM3 IPGCM4 IPGCM5 IPGCM6 IPGCM7 IPGCMS IPGCPT IPGTM ISM0 MK0 PB0 TM0LW TM0UW TM1 TM2						

2.4 Label Generation Rule

When expanding a control statement into assembler instruction(s), the structured assembler generates a label for a branch instruction.

Because each label generated by the structured assembler begins with "??", do not use a label which begins with "?" in the source program.

CHAPTER 3. CONTROL STATEMENTS

3.1 Overview of Control Statements

Control statements are used to structurally describe the flow of program control and are divided into the following two types:

- o Conditional branching (IF-THEN-ELSE type)
  - (1) if-elseif-else-endif
  - (2) if_bit-elseif_bit-else-endif
  - (3) switch-case-default-ends
- o Condition loop (DO-WHILE type)
  - (4) for-next (repeat of specified increments)
  - (5) while-endw (repeat of condition judgment before processing)
  - (6) while_bit-endw (repeat of condition judgment before processing)
  - (7) repeat-until (repeat of condition judgment after processing)
  - (8) repeat-until_bit (repeat of condition judgment after processing)
  - (9) continue (repeat of loop block)
  - (10) break (exit from loop block)
  - (11) goto (exit for exception processing)

3.2 Characters in Control Statements

With the conditional branch instructions of the 78K series microcomputers, branching is allowed only for relative addresses of up to 128 bytes (about 40 instructions). Because a control statement generates a conditional branch instruction, use uppercase or lowercase letters for the control statement to specify whether or not its relative address is within 128 bytes.

"if" and "elseif" in control statements are reserved words. The structured assembler makes judgment on whether the control statement has been described in uppercase or lowercase letters according to the first character (uppercase or lowercase letter) of such reserved word in the control statement.

IF, If ... The statement begins with the uppercase letter "I" and thus is judged to have been described in uppercase letters.

if, iF ... The statement begins with the lowercase letter "i" and thus is judged to have been described in lowercase letters.

If the statement is described in uppercase letters .....
Branching is executed based on a combination of a conditional branch instruction and a BR directive.

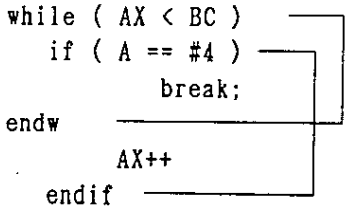
If the statement is described in lowercase letters ....
Branching is executed directly with a conditional directive.

In control statements which must be paired (e.g., if, else, and endif), uppercase and lowercase letters may be mixed for statement description. In other words, the statements may be described such as "IF-else-ENDIF".

3.3 Nesting

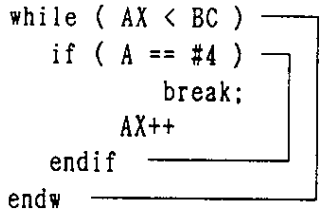
Control statements can be nested up to 31 levels. However, no two control statements can be described across each other.

(Example of incorrect description)



Error results because "if-endif" is described across "while-endw".

(Example of correct description)



Correct nesting because "if" statement is described within "while" statement.

3.4 Functions of Control Statements

The function of each control statement is explained in this section.

(1) if-elseif-else-endif

[Description format]

```
if (condition expression 1) [(register name)]
    if clause
elseif (condition expression 2) [(register name)]
    elseif clause
else
    else clause
endif
```

[Function]

① **if-endif**

If the condition specified by condition expression 1 is True, the "if" clause will be executed.
The "if" clause may consist of two or more lines.

② **if-else-endif**

If the condition specified by condition expression 1 is True, the "if" clause will be executed. If it is False, the "else" clause will be executed.
The "if" and "else" clauses each may consist of two or more lines.

③ **if-elseif-else-endif**

Two or more **elseif** statements may be described for one **if** statement.
If the condition specified by condition expression 1 is True, the "if" clause will be executed. If it is False, condition expression 2 will be judged for True/False.
If the condition specified by condition expression 2 is True, the "elseif" clause will be executed. If it is False and if another **elseif** statement exists before the **endif** statement, the condition specified by the **elseif** statement will be judged. If no other **elseif** statement exists, the "else" clause will be executed.

The "if", "elseif", and "else" clauses each may consist of two or more lines.

④ Uppercase and lowercase letters
Instructions to be generated when the **if** or **elseif** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an "if", "elseif", or "else" clause is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an "if", "elseif", or "else" clause is relatively short, that is, when the number of bytes between **if** and **elseif**, between **elseif** and **else**, or between **else** and **endif** statements does not exceed 128 bytes.

[Explanation]

① A relative operator expression or logical operator expression must be described as condition expression 1 or 2. See 3.5, Condition Expressions for the relative operator and logical operator expressions.

② The **if-else-endif** statement is described when branching conditionally to two points.

③ The **if-elseif-else-endif** statement is described when branching to multiple points with a value range. This statement is different from the **switch** statement in that it can have a range of values.

④ If a register name is specified following condition expression 1 or 2, the condition specified by the expression is judged for True/False by using the specified register.

⑤ The **elseif** or **else** statement may be omitted. Two or more **elseif** statements may be described for one **if** statement.

[Instructions to be generated]

① Processing of "**if** (condition expression)"

(1) Generates an instruction for the True/False condition judgment made on the condition expression.

(2) Generates a branch instruction to the "elseif" or "else" clause if the condition is not met.
If the if statement is described in uppercase letters (described as IF or If), a 3-byte branch instruction is generated.

② Processing of "**elseif** (condition expression)"

(1) Generates a branch instruction to the **endif** statement.

(2) Generates a label for the branch instruction to be generated by the **if** statement.

(3) Generates an instruction for the True/False condition judgment made on the condition expression.

(4) Generates a branch instruction to the "elseif" or "else" clause if the condition is not met.
If the if statement is described in uppercase letters (described as IF or If), a 3-byte branch instruction is generated.

③ Processing of "**else**"

(1) Generates a branch instruction to the **endif** statement.

(2) Generates a label for the branch instruction to be generated by the **if** or **elseif** statement.

④ Processing of "**endif**"

Generates a label for the branch instruction to be generated by the **if**, **elseif**, or **else** statement.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
if ( A == #0 )
   TMC0.7 = TFLG.0 ( CY )
   BC = TM0 ( AX )
else
   BC = #0A00H
endif
```

<Output source program>

```
;   if ( A == #0 )
                                    CMP     A,#0
                                    BNZ     $??1
;       TMC0.7 = TFLG.0 ( CY )
                                    MOV1    CY,TFLG.0
                                    MOV1    TMC0.7,CY
;       BC = TM0 ( AX )
                                    MOVW    AX,TM0
                                    MOVW    BC,AX
;   else
                                    BR      ??2
                              ??1:
;       BC = #0A00H
                                    MOVW    BC,#0A00H
;   endif
                              ??2:
```

(b) When the statements are described in uppercase letters

<Input source program>

```
IF ( A == #0 )
   TMC0.7 = TFLG.0 ( CY )
   BC = TM0 ( AX )
ELSE
   BC = #0A00H
ENDIF
```

<Output source program>

```
;   IF ( A == #0 )
                                CMP     A,#0
                                BZ      $??1
                                BR      ??2
                          ??1:
;       TMC0.7 = TFLG.0 ( CY )
                                MOV1    CY,TFLG.0
                                MOV1    TMC0.7,CY
;       BC = TM0 ( AX )
                                MOVW    AX,TM0
                                MOVW    BC,AX
;   ELSE
                                BR      ??3
                          ??2:
;       BC = #0A00H
                                MOVW    BC,#0A00H
;   ENDIF
                          ??3:
```

(2) if_bit-elseif_bit-else-endif

[Description format]

```
if_bit (bit condition expression 1)
    if clause
elseif_bit (bit condition expression 2)
    elseif clause
else
    else clause
endif
```

[Function]

① **if_bit-endif**

If the condition specified by bit condition expression 1 is True, the "if" clause will be executed.
The "if" clause may consist of two or more lines.

② **if_bit-else-endif**

If the condition specified by bit condition expression 1 is True, the "if" clause will be executed. If it is False, the "else" cause will be executed.
The "if" and "else" clauses each may consist of two or more lines.

③ **if_bit-elseif_bit-else-endif**

If the condition specified by bit condition expression 1 is True, the "if" clause will be executed. If it is False, bit condition expression 2 will be judged for True/False.
If the condition specified by bit condition expression 2 is True, the "elseif" clause will be executed. If it is False and if another **elseif_bit** statement exists before the **endif** statement, the condition specified by the **elseif_bit** statement will be judged. If no other **elseif_bit** statement exists, the "else" clause will be executed.
The "if", "elseif", and "else" clauses each may consist of two or more lines.

④ Uppercase and lowercase letters
Instructions to be generated when the **if_bit** or **elseif_bit** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an "if", "elseif", or "else" clause is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an "if", "elseif", or "else" clause is relatively short, that is, when the number of bytes between **if_bit** and **elseif_bit**, between **elseif_bit** and **else**, or between **else** and **endif** statements does not exceed 128 bytes.

[Explanation]

① A bit condition expression must be described as bit condition expression 1 or 2.
See 3.5, Condition Expressions for the bit condition expressions.

② The **if_bit-else-endif** statement is described when branching conditionally to two points.
The **if_bit-elseif_bit-else-endif** statement is described when branching to multiple points by checking two or more bit symbols.

③ The **elseif_bit** or **else** statement may be omitted. Two or more **elseif_bit** statements may be described for one **if_bit** statement.

[Instructions to be generated]

① Processing of "**if_bit** (bit condition expression)"
Generates an instruction for the True/False condition judgment made on the bit condition expression.

② Processing of "**elseif_bit** (bit condition expression)"
(1) Generates a label for the branch instruction to be generated by the **if_bit** statement.
(2) Generates an instruction for the True/False condition judgment made on the bit condition expression.

③ Processing of "**else**"
(1) Generates a branch instruction to the **endif** statement.
(2) Generates a label for the branch instruction to be generated by the **if_bit** or **elseif_bit** statement.

④ Processing of "**endif**"
Generates a label for the branch instruction to be generated by the **if_bit**, **elseif_bit**, or **else** statement.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
if_bit ( !TRFG.0 )
  !CNT = #0EH (A)
  SET1  PRTYFLG.3
elseif_bit ( PGF.0 )
  SIO = RDATA (AX)
  BC = #0FFCH
 else
  H = #(FG SHR 6)
  CY = PFG.0
  CLR1  BUSYFG.2
endif
```

<Output source program>

```
;   if_bit ( !TRFG.0 )
                                BT      TRFG.0,$??1
;     !CNT = #0EH (A)
                                MOV     A,#0EH
                                MOV     !CNT,A
      SET1  PRTYFLG.3
;   elseif_bit ( PGF.0 )
                          ??1:
                                BF      PGF.0,$??2
;     SIO = RDATA (AX)
                                MOVW    AX,RDATA
                                MOVW    SIO,AX
;     BC = #0FFCH
                                MOVW    BC,#0FFCH
;    else
                                BR      ??3
                          ??2:
;     H = #(FG SHR 6)
                                MOV     H,#(FG SHR 6)
;     CY = PFG.0
                                MOV1    CY,PFG.0
      CLR1  BUSYFG.2
;    endif
                          ??3:
```

(b) When the statements are described in uppercase letters

<Input source program>

```
IF_BIT ( !TRFG.0 )
  !CNT = #0EH (A)
  SET1  PRTYFLG.3
ELSEIF_BIT ( PGF.0 )
  SIO = RDATA (AX)
  BC = #0FFCH
ELSE
  H = #(FG SHR 6)
  CY = PFG.0
  CLR1  BUSYFG.2
ENDIF
```

<Output source program>

```
;   IF_BIT ( !TRFG.0 )
                                BF      TRFG.0,$??1
                                BR      ??2
                          ??1:
;     !CNT = #0EH (A)
                                MOV     A,#0EH
                                MOV     !CNT,A
      SET1  PRTYFLG.3
;   ELSEIF_BIT ( PGF.0 )
                          ??2:
                                BT      PGF.0,$??3
                                BR      ??4
                          ??3:
;     SIO = RDATA (AX)
                                MOVW    AX,RDATA
                                MOVW    SIO,AX
;     BC = #0FFCH
                                MOVW     BC,#0FFCH
;    ELSE
                                BR      ??5
                          ??4:
;     H = #(FG SHR 6)
                                MOV     H,#(FG SHR 6)
;     CY = PFG.0
                                MOV1    CY,PFG.0
      CLR1  BUSYFG.2
;   ENDIF
                          ??5:
```

(3) switch-case-default-ends

[Description format]

```
switch (α)
    case   constant 1:
                        case 1 clause
    case   constant 2:
                        case 2 clause
              :
    case   constant N:
                        case N clause

    default:
                        default clause
ends
```

[Function]

① **case**

For the constant i (where i= 1 to N) which coincides with the value of α , the "case i" clause will be executed.

② **default**

If the value of α does not coincide with the constant 1 to N, the "default" clause will be executed.

The **default** statement and "default" clause may be omitted from description.

③ Uppercase and lowercase letters

Instructions to be generated when the **case** statements are described in uppercase letters are different from those when the statements are described in lowercase letters.

Uppercase letters ... Used when a "case" clause is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when a "case" clause is relatively short, that is, when the number of bytes between one case statement and the next does not exceed 128 bytes.

[Explanation]

① For α (alpha) that can be described in the **switch** statement, see Appendix A, "List of Statement Structures".

② Even if a "case i" clause is executed as a result of agreement of the value between "α" and constant i, other "case" clauses in subsequent **case** statements will also be executed if the value of α coincides with constant i. Therefore, to execute only one "case" clause per **switch** statement, a **break** statement must be described at the end of each "case" clause. (See Application examples below.)

③ As constant i, any of numeric constants (binary, octal, decimal, and hexadecimal) and character constants may be described. However, because this structured assembler processor also recognizes constants as character strings, the specified constant must be the one which can be interpreted as a constant by the assembler.

④ The contents (value) of the A or AX register are subject to change. Therefore, if the value of the A or AX register must be retained, save the value of the register before executing the **switch** statement.

[Instructions to be generated]

① Processing of "**switch** (α)"

Generates the following instruction if the value of α is not "A" or "AX" register.

MOV A, α or MOVW AX, α

② Processing of "**case**"

(1) Generates a label for the branch instruction to be generated by the preceding **case** statement.

(2) Generates the following two instructions and branches to the next **case**, **default**, or **ends** statement if the value of α does not coincide with the constant "i".

CMP A,constant i or CMPW AX, constant i

BNZ ??false

③ Processing of "**default**"

Generates a label for the branch instruction to be generated by the **case** statement.

④ Processing of "**ends**"

Generates a label for the branch instruction to be generated by the **case** or **break** statement.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
switch ( MODEP )
  case 1:
      if_bit ( PORT.0 )
          SET1  BTM.3
      endif
      BC = TM0 (AX)
      break
  case 2:
      BC = #0A00H
      break
  case 3:
      BC = #0A000H
      break
  default:
      BC = #0000H
ends
```

<Output source program>

```
;   switch ( MODEP )
                              MOVW    AX,MODEP
;    case 1:
                        ??1:
                              CMPW    AX,#1
                              BNZ     $??2
;       if_bit ( PORT.0 )
                              BF      PORT.0,$??3
          SET1  BTM.3
;       endif
                        ??3:
;       BC = TM0 (AX)
                              MOVW    AX,TM0
                              MOVW    BC,AX
;       break
                              BR      ??4
;    case 2:
                        ??2:
                              CMPW    AX,#2
                              BNZ     $??5
;       BC = #0A00H
                              MOVW    BC,#0A00H
;       break
                              BR      ??4
;    case 3:
                        ??5:
                              CMPW    AX,#3
                              BNZ     $??6
;       BC = #0A000H
                              MOVW    BC,#0A000H
;       break
                              BR      ??4
;    default:
                        ??6:
;       BC = #0000H
                              MOVW    BC,#0000H
;  ends
                        ??4:
```

(b) When the statements are described in uppercase letters

<Input source program>

```
SWITCH( MODEP )
  CASE 1:
     if_bit ( PORT1.0 )
        SET1  BTM.3
     endif
     BC = TM0 (AX)
     break
  CASE 2:
     BC = #0A00H
     break
  CASE 3:
     BC = #0A000H
     break
  DEFAULT:
     BC = #0000H
ENDS
```

<Output source program>

```
;   SWITCH( MODEP )
                            MOVW    AX,MODEP
;     CASE 1:
                    ??1:
                            CMPW    AX,#1
                            BZ      $??2
                            BR      ??3
                    ??2:
;         if_bit ( PORT1.0 )
                            BF      PORT.0,$??4
              SET1  BTM.3
;         endif
                    ??4:
;         BC = TM0 (AX)
                            MOVW    AX,TM0
                            MOVW    BC,AX
;         break
                            BR      ??5
;     CASE 2:
                    ??3:
                            CMPW    AX,#2
                            BZ      $??6
                            BR      ??7
                    ??6:
;         BC = #0A00H
                            MOVW    BC,#0A00H
;         break
                            BR      ??5
;     CASE 3:
                    ??7:
                            CMPW    AX,#3
                            BZ      $??8
                            BR      ??9
                    ??8:
;         BC = #0A000H
                            MOVW    BC,#0A000H
;         break
                            BR      ??5
;     DEFAULT:
                    ??9:
;         BC = #0000H
                            MOVW    BC,#0000H
;   ENDS
                    ??5:
```

(4) for-next

[Description format]

```
for (expression 1;expression 2;expression 3) [( α )]
   instruction group
next
```

[Function]

① Executes a group of instructions repeatedly while the condition specified by a parameter is being met.

expression 1 ... Sets the initial value of the parameter (for example, β = #0).

expression 2 ... Sets the repeat condition (for example, β < #7FH).

expression 3 ... Specifies the increment or decrement operation of the counter value.

② If (α) is specified, a range of values that can be specified in expression 1, expression 2, or expression 3 is widened. The alpha (α) is used for work by the **for** statement.

③ Uppercase and lowercase letters
Instructions to be generated when the **for** or **next** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an instruction group is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an instruction group is relatively short and the sum of the number of bytes of the instructions expanded by the for statement and the instruction group does not exceed 128 bytes.

[Explanation]

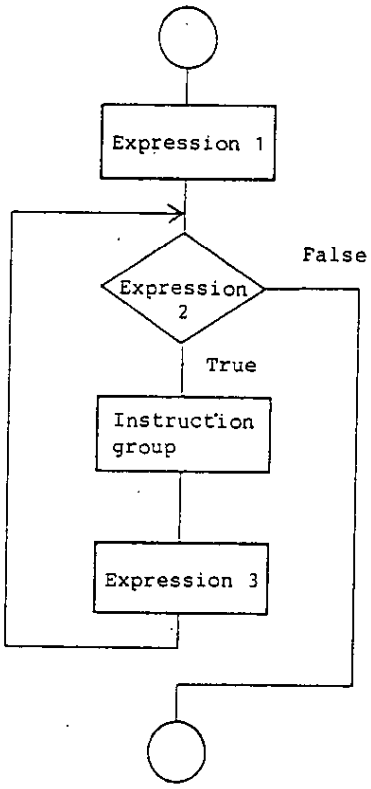
① Expressions

An expression statement must be described in expression 1 as the initial value of a parameter. For example, $\beta = \gamma$.

A relational operator expression or logical operator expression may be described in expression 2 as the repeat condition. For example, $\beta < \delta$, β && δ .

An expression statement must be described in expression 3. For example, β ++ , β --.

② The operation of a **for** loop is as shown below.



③ For α (alpha), β (beta), γ (gamma), and δ (delta) that can be described in the **for** statement, see Appendix A, "List of Statement Structures".

④ expression 2 and expression 3 function to control the **for** block and thus the contents of each expression cannot be changed with an executable statement. If changed, the processor may malfunction.

⑤ The **for-next** block can be rewritten with a **while** statement as follows:

```
expression 1
while (expression 2)
  instruction group
  expression 3
endw
```

Instructions to be generated in this case will differ from those when the **for-next** block is used.

⑥ Use of "saddr" data instead of register values as the control variables of expression 1 and expression 3 will result in generation of a better code.
If a register value is to be specified as a control variable, use the A or AX register.

⑦ In the **for** statement, execution of an instruction group can be repeated up to 255 times. If the instruction group is to be repeated 256 times or more, nest the **for** statement and use two "saddr" data as the control variables.

[Instructions to be generated]

① Processing of "**for** (expression 1;expression 2:expression 3)"
(1) Generates an instruction for expression 1. If a register name is specified, instructions are generated using the specified register (for substitution or relational operation).
(2) Generates a branch instruction to the statement which makes the True/False judgment on the condition specified by expression 2.
(3) Generates a label for the branch instruction to be generated by the **next** statement.
(4) Generates an instruction for expression 3.
(5) Generates a label for the branch instruction to be generated in (2) above.
(6) Generates an instruction to make the True/False judgment of the condition specified by expression 2.

② Processing of "**next**"
(1) Generates a branch instruction to the label generated in (3) above in the **for** statement processing.
(2) Generates a label for a branch instruction to exit from the **for** statement block.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
; i is saddr, used for a counter.

    for ( i = #0H; i < #0FFH; i++ )
      CALL  !XXX
    next
```

<Output source program>

```
; i is saddr, used for a counter.

;   for ( i = #0H; i < #0FFH; i++ )
                                MOV     i,#0H
                                BR      ??1
                            ??2:
                                INC     i
                            ??1:
                                CMP     i,#0FFH
                                BNC     $??3
      CALL  !XXX
;   next
                                BR      ??2
                            ??3:
```

(b) When the statements are described in uppercase letters

<Input source program>

```
FOR ( i = #0H; i < #0FFH; i++ )
  CALL  !XXX
NEXT
```

<Output source program>

```
;    FOR ( i = #0H; i < #0FFH; i++ )
                              MOV     i,#0H
                              BR      ??1
                         ??2:
                              INC     i
                         ??1:
                              CMP     i,#0FFH
                              BC      $??3
                              BR      ??4
                         ??3:
     CALL  !XXX
;   NEXT
                              BR      ??2
                         ??4:
```

(5) while-endw

[Description format]

```
while (condition expression) [( α )]
   instruction group
endw
```

[Function]

① Executes a group of instructions repeatedly while the condition specified by condition expression is True.

② Uppercase and lowercase letters
Instructions to be generated when the **while** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an instruction group is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an instruction group is relatively short, that is, when the sum of the number of bytes of the instructions expanded by the **while** statement and the instruction group does not exceed 128 bytes.

[Explanation]

① As a condition expression, a relational operator expression, logical operator expression, or "forever" may be described. If "forever" is described, an endless loop will result.

② As the value of α (alpha), a register name to be used for the described relational operator or logical operator expression must be specified.

③ Because the condition expression will be evaluated before executing a group of instructions, the instruction group will not be executed at all if the condition is False to begin with.

[Instructions to be generated]

① Processing of "**while** (condition expression)"

(1) Generates a label for the branch instruction to be generated by the **endw** statement.

(2) Generates an instruction to make the True/False judgment on the condition specified by the condition expression. If a register name is specified, the True/False judgment instruction will be generated using by the specified register.

(3) Generates a branch instruction to exit from the **while** statement block if the result of the condition judgment is False.

② Processing of "**endw**"

(1) Generates a branch instruction for repeating the execution of the instruction group.

(2) Generates a label for the branch instruction to exit from the **while** statement block.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
while ( CNTP < #0FFFH ) (AX)
   if ( MEM == #0 )
      break
   endif
   HL++
endw
```

<Output source program>

```
;   while ( CNTP < #0FFFH ) (AX)
                          ??1:
                                MOVW  AX, CNTP
                                CMPW  AX, #0FFFH
                                BNC   $??2
;      if ( MEM == #0 )
                                CMP    MEM, #0
                                BNZ    $??3
;         break
                                BR     ??2
;      endif
                          ??3:
;      HL++
                                INCW   HL
;   endw
                                BR     ??1
                          ??2:
```

(b) When the statements are described in uppercase letters

<Input source program>

```
WHILE ( CNTP < #0FFFH ) (AX)
   if ( MEM == #0 )
      break
   endif
   HL++
ENDW
```

<Output source program>

```
;  WHILE ( AX < #0FFFH )
                        ??1:
                              CMPW   AX,#0FFFH
                              BC     $??2
                              BR     ??3
                        ??2:
;      if ( MEM == #0 )
                              CMP    MEM,#0
                              BNZ    $??4
;         break
                              BR     ??3
;      endif
                        ??4:
;      HL++
                              INCW   HL
;  ENDW
                              BR     ??1
                        ??3:
```

(6) **while_bit-endw**

[Description format]

```
while_bit (bit condition expression)
   instruction group
endw
```

[Function]

① Executes a group of instructions repeatedly while the condition specified by bit condition expression is True.

② Uppercase and lowercase letters
Instructions to be generated when the **while_bit** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an instruction group is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an instruction group is relatively short, that is, when the sum of the number of bytes of the instructions expanded by the **while_bit** statement and the instruction group does not exceed 128 bytes.

[Explanation]

Because the bit condition expression will be evaluated before executing a group of instructions, the instruction group will not be executed at all if the bit condition is False to begin with.

[Instructions to be generated]

① Processing of "**while_bit** (bit condition expression)"

(1) Generates a label for the branch instruction to be generated by the **endw** statement.

(2) Generates an instruction to make the True/False judgment on the condition specified by the bit condition expression.

(3) Generates a branch instruction to exit from the **while_bit** statement block.

② Processing of "**endw**"

(1) Generates a branch instruction for repeating the execution of the instruction group.

(2) Generates a label for the branch instruction to exit from the **while_bit** statement block.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
while_bit ( !TRFG.0 )
     A  = PORT1
     if ( A == #4H )
       X  = #0FFH
     else
       CLR1  PFG.0
     endif
endw
```

<Output source program>

```
;   while_bit ( !TRFG.0 )
                        ??1:
                                BT      TRFG.0,$???2
;       A  = PORT1
                                MOV     A,PORT1
;       if ( A == #4H )
                                CMP     A,#04H
                                BNZ     $???3
;          X  = #0FFH
                                MOV     X,#0FFH
;       else
                                BR      ???4
                        ???3:
          CLR1  PFG.0
;       endif
                        ???4:
;   endw
                                BR      ??1
                        ???2:
```

(b) When the statements are described in uppercase letters

<Input source program>

```
WHILE_BIT ( !TRFG.0 )
     A  = PORT1
     if ( A == #4H )
       X  = #0FFH
     else
       CLR1  PFG.0
     endif
ENDW
```

<Output source program>

```
;   WHILE_BIT ( !TRFG.0 )
                        ??1:
                              BF      TRFG.0,$??2
                              BR      ??3
                        ??2:
;        A  = PORT1
                              MOV     A,PORT1
;        if ( A == #4H )
                              CMP     A,#04H
                              BNZ     $??4
;          X  = #0FFH
                              MOV     X,#0FFH
;        else
                              BR      ??5
                        ??4:
           CLR1  PFG.0
;        endif
                        ??5:
;   ENDW
                              BR      ??1
                        ??3:
```

(7) repeat-until

[Description format]

```
repeat
   instruction group
until (condition expression)  [( α )]
```

[Function]

① Executes a group of instructions repeatedly while the condition specified by condition expression is False.

② Uppercase and lowercase letters
Instructions to be generated when the **until** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an instruction group is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an instruction group is relatively short, that is, when the sum of the number of bytes of the instructions expanded by the **until** statement and the instruction group does not exceed 128 bytes.

[Explanation]

① As a condition expression, a relational operator expression, logical operator expression, or "forever" may be described. If "forever" is described, an endless loop will result.

② As the value of α (alpha), a register name to be used for the described relational operator or logical operator expression must be specified.

③ Because the condition expression will be evaluated after executing a group of instructions, the instruction group will be executed once even if the condition is True to begin with.

[Instructions to be generated]

① Processing of "**repeat**"
Generates a label for the branch instruction to be generated by the **until** statement.

② Processing of "**until** (condition expression)"
(1) Generates an instruction to make the True/False judgment on the condition specified by the condition expression.
(2) Generates a branch instruction for repeating the execution of the instruction group.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
repeat
   BC = DE
   if ( ABC != #0CH )
       CALL  !XXX
   endif
   CNT++
until ( CNT == #0FFH )
```

<Output source program>

```
;   repeat
                        ??1:
;      BC = DE
                                MOVW    BC, DE
;      if ( ABC != #0CH )
                                CMP     ABC, #0CH
                                BZ      $??2
          CALL  !XXX
;      endif
                        ??2:
;      CNT++
                                INC     CNT
;    until ( CNT == #0FFH )
                                CMP     CNT, #0FFH
                                BNZ     $??1
```

(b) When the statements are described in uppercase letters

<Input source program>

```
REPEAT
   BC = DE
   if ( ABC != #0CH )
       CALL  !XXX
   endif
   CNT++
UNTIL ( CNT == #0FFH )
```

<Output source program>

```
;  REPEAT
                        ??1:
;     BC = DE
                                MOVW    BC, DE
;     if ( ABC != #0CH )
                                CMP     ABC, #0CH
                                BZ      $??2
         CALL  !XXX
;     endif
                        ??2:
;     CNT++
                                INC     CNT
;   UNTIL ( CNT == #0FFH )
                                CMP     CNT, #0FFH
                                BZ      $??3
                                BR      $??1
                        ??3:
```

(8) repeat-until_bit

[Description format]

```
repeat
   instruction group
until_bit (bit condition expression)
```

[Function]

① Executes a group of instructions repeatedly while the condition specified by bit condition expression is False.

② Uppercase and lowercase letters
Instructions to be generated when the **until_bit** statement is described in uppercase letters are different from those when the statement is described in lowercase letters.

Uppercase letters ... Used when an instruction group is extremely long. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ... Used when an instruction group is relatively short, that is, when the sum of the number of bytes of the instructions expanded by the **until_bit** statement and the instruction group does not exceed 128 bytes.

[Explanation]

Because the bit condition expression will be evaluated after executing a group of instructions, the instruction group will be executed once even if the condition is True to begin with.

[Instructions to be generated]

① Processing of "**repeat**"

Generates a label for the branch instruction to be generated by the **until_bit** statement.

② Processing of "**until_bit** (bit condition expression)"

(1) Generates an instruction to make the True/False judgment on the condition specified by the bit condition expression.

(2) Generates a branch instruction for repeating the execution of the instruction group.

[Application examples]

(a) When the statements are described in lowercase letters

<Input source program>

```
repeat
   CR00 = AX
   CALL !xxx
   AX = TM0
until_bit(  TRIGF.0  )
```

<Output source program>

```
;  repeat
                        ??1:
;     CR00 = AX
                              MOVW    CR00,AX
      CALL !xxx
;     AX = TM0
                              MOVW    AX,TM0
;  until_bit(  TRIGF.0  )
                              BF      TRIGF.0,$??1
```

(b) When the statements are described in uppercase letters

<Input source program>

```
REPEAT
   CR00 = AX
   CALL !xxx
   AX = TM0
UNTIL_BIT(  TRIGF.0  )
```

<Output source program>

```
;  REPEAT
                        ??1:
;     CR00 = AX
                              MOVW    CR00,AX
      CALL !xxx
;     AX = TM0
                              MOVW    AX,TM0
;  UNTIL_BIT(  TRIGF.0  )
                              BT      TRIGF.0,$??2
                              BR      ??1
                        ??2:
```

(9) break

[Description format]

```
break
```

[Function]

Terminates the innermost **while**, **repeat**, **for**, or **switch** statement block, regardless of how these statements might be nested.

[Explanation]

A **break** statement cannot be described in other than **while**, **repeat**, **for**, and **switch** statement blocks. If described, an error will result.

[Instructions to be generated]

Generates a branch instruction to exit from the **while**, **repeat**, **for**, or **switch** statement block.

[Application example]

<Input source program>

```
while ( forever )
   A = #0
   PORT4 = A
   if ( A == #0FH )
       break
   endif
   HL++
endw
```

<Output source program>

```
;    while ( forever )
                          ??1:
;       A = #0
                                MOV     A,#0
;       PORT4 = A
                                MOV     PORT4,A
;       if ( A == #0FH )
                                CMP     A,#0FH
                                BNZ     ??2
;           break
                                BR      ??3
;       endif
                          ??2:
;       HL++
                                INCW    HL
;    endw
                                BR      ??1
                          ??3:
```

(10) continue

[Description format]

```
continue
```

[Function]

Causes unconditional branching to the first label of the innermost **while**, **repeat**, or **for** statement in a loop and executes the next loop.

[Explanation]

① **continue** is used to bypass code in a loop (to skip the subsequent processing in a block under execution) and to force the next loop to be repeated.

② A **continue** statement cannot be described in other than the **while**, **repeat**, and **for** statement blocks. If described, an error will result.

[Instructions to be generated]

Generates a branch instruction to the label for repeating the **while**, **repeat**, or **for** loop.

[Application example]

<Input source program>

```
while ( forever )
   A = #0
   PORT4 = A
   if ( A == #0FH )
       break
   elseif ( A == #0CH )
       continue
   else
   HL++
   endif
endw
```

<Output source program>

```
;    while ( forever )
                          ??1:
;       A = #0
                                MOV     A,#0
;       PORT4 = A
                                MOV     PORT4,A
;       if ( A == #0FH )
                                CMP     A,#0FH
                                BNZ     ??2
;           break
                                BR      ??3
;       elseif ( A == #0CH )
                                BR      ??4
                          ??2:
                                CMP     A,#0CH
                                BNZ     ??5
;           continue
                                BR      ??1
;       else
                                BR      ??4
                          ??5:
;       HL++
                                INCW    HL
;       endif
                          ??4:
;      endw
                                BR      ??1
                          ??3:
```

(11) goto

[Description format]

```
goto   label
```

[Function]

Causes program execution to unconditionally jump to the specified label.

[Explanation]

① **goto** is used when an error handling is immediately required in a program to process errors or when a common processing procedure can be applied to an error which has occurred at two or more locations.

② As "label", a symbol which has been described in the LABEL field of the assembly language must be specified.

[Instructions to be generated]

Generates the following instruction:

```
BR     label
```

[Application example]

<Input source program>

```
while ( forever )
   A = #0
   PORT4 = A
   if ( A == #0FH )
       goto ERROR
   endif
   HL++
endw
```

<Output source program>

```
;    while ( forever )
                          ??1:
;       A = #0
                                MOV     A,#0
;       PORT4 = A
                                MOV     PORT4,A
;       if ( A == #0FH )
                                CMP     A,#0FH
                                BNZ     ??2
;           goto ERROR
                                BR      ERROR
;       endif
                          ??2:
;       HL++
                                INCW    HL
;    endw
                                BR      ??1
                          ??3:
```

3.5 Condition Expressions

Condition expressions are used to set conditions in a control statement and are available in the following three types:

- o Relational operator expression
  Compares the value of the first (left) term of the expression with that of the second (right) term for True or False determination.
- o Bit condition expression
  Determines the ON or OFF state of a flag according to the value of a specified bit symbol.
- o Logical operator expression
  Performs a logical operation on two specified condition expressions when two conditions are to be combined.

Relational operator expression		Description format	Function
(1)	Equal	$\alpha == \beta$	True if $\alpha = \beta$; False if $\alpha \neq \beta$.
(2)	Not Equal	$\alpha != \beta$	True if $\alpha \neq \beta$; False if $\alpha = \beta$.
(3)	Less Than	$\alpha < \beta$	True if $\alpha < \beta$; False if $\alpha >= \beta$.
(4)	Greater Than	$\alpha > \beta$	True if $\alpha > \beta$; False if $\alpha <= \beta$.
(5)	Greater Than/ Equal	$\alpha >= \beta$	True if $\alpha >= \beta$; False if $\alpha < \beta$.
(6)	Less Than/ Equal	$\alpha <= \beta$	True if $\alpha <= \beta$; False if $\alpha > \beta$.

Bit condition expression		Description format	Function
(7)	Positive logic (bit)	bit symbol	True if specified bit is 1; otherwise, False.
(8)	Negative logic (bit)	!bit symbol	True if specified bit is 0; otherwise, False.

Logical operator expression		Description format	Function
(9)	AND	expression 1 && expression 2	True if both expression 1 and expression 2 are True.
(10)	OR	expression 1 || expression 2	True if either expression 1 or expression 2 is True.

By specifying γ (gamma) after a relational operator expression, two values, α (alpha) and β (beta), which cannot be compared directly with each other, can be compared. The γ (gamma) specifies a register name which is to be destroyed for the indirect comparison of the two values.

(1) Equal (==)

[Description format]

α == β [(γ)] γ : register name

[Function]

① If the value of the left term "α"(alpha) is equal to the value of the right term "β"(beta), True ("1") is returned; otherwise, False ("0") is returned.

② If γ (register name) is specified, the value of α (alpha) is transferred to the register specified by γ (gamma). If the value of γ (gamma) is equal to the value of β (beta), True is returned; otherwise, False is returned.

③ Uppercase and lowercase letters
Instructions to be generated will differ depending on whether the control statement in which the relational operator expression is specified has been described in uppercase or lowercase letters.

Uppercase letters ..... Used when a jump-to-address by the control statement is extremely distant. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ..... Used when a jump-to-address by the control statement are within a range of 128 bytes.

[Explanation]

For α (alpha), β (beta), and γ (gamma) that can be described as the operands of this relational operator expression, see Appendix B, "List of Operands".

[Instructions to be generated]

① If the control statement is described in lowercase letters and no register name is specified, the following instructions are generated:

```
CMP (W)  α, β
BNZ      $??FALSE
```

② If the control statement is described in lowercase letters and a register name is specified, the following instructions are generated:

```
MOV (W)  register, α
CMP (W)  register, β
BNZ      $??FALSE
```

③ If the control statement is described in uppercase letters and no register name is specified, the following instructions are generated:

```
CMP (W)  α, β
BZ       $??TRUE
BR       ??FALSE
??TRUE:
```

④ If the control statement is described in uppercase letters and a register name is specified, the following instructions are generated:

```
MOV (W)  register, α
CMP (W)  register, β
BZ       $??TRUE
BR       ??FALSE
??TRUE:
```

[Application example]

<Input source program>

```
    if ( AX == HL )
        CALL    !XXX
    else
        CALL    !YYY
    endif
```

```
    if ( !ABC == #5 )  ( A )
        CALL    !PPP
    endif
```

```
;   IF ( AX == HL )
        CALL    !XXX
;   ELSE
        CALL    !YYY
;   ENDIF
```

```
;   IF ( !ABC == #5 )  ( A )
        CALL    !PPP
;   ENDIF
```

<Output source program>

```
;  if ( AX == HL )
                        CMPW    AX, HL
                        BNZ     $??1
        CALL    !XXX
;  else
                        BR      ??2
                ??1:
        CALL    !YYY
;  endif
                ??2:
```

```
;  if ( !ABC == #5 )  ( A )
                        MOV     A, !ABC
                        CMP     A, #5
                        BNZ     $??3
        CALL    !PPP
;  endif
                ??3:
```

```
;  IF ( AX == HL )
                        CMPW    AX, HL
                        BZ      $??4
                        BR      ??5
                ??4:
        CALL    !XXX
;  ELSE
                        BR      ??6
                ??5:
        CALL    !YYY
;  ENDIF
                ??6:
```

```
;  IF ( !ABC == #5 )  ( A )
                        MOV     A, !ABC
                        CMP     A, #5
                        BZ      $??7
                        BR      ??8
                ??7:
        CALL    !PPP
;  ENDIF
                ??8:
```

(2) Not Equal (!=)

[Description format]

α != β [(γ)] γ : register name

[Function]

① If the value of the left term "α"(alpha) is not equal to the value of the right term "β"(beta), True ("1") is returned; otherwise, False ("0") is returned.

② If γ (register name) is specified, the value of α (alpha) is transferred to the register specified by γ (gamma). If the value of γ (gamma) is not equal to the value of β (beta), True is returned; otherwise, False is returned.

③ Uppercase and lowercase letters
Instructions to be generated will differ depending on whether the control statement in which the relational operator expression is specified has been described in uppercase or lowercase letters.

Uppercase letters ..... Used when a jump-to-address by the control statement is extremely distant. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ..... Used when a jump-to-address by the control statement is within a range of 128 bytes.

[Explanation]

For α (alpha), β (beta), and γ (gamma) that can be described in this relational operator expression, see Appendix B, "List of Operands".

[Instructions to be generated]

① If the control statement is described in lowercase letters and no register name is specified, the following instructions are generated:

```
CMP (W)  α, β
BZ       $??FALSE
```

② If the control statement is described in lowercase letters and a register name is specified, the following instructions are generated:

```
MOV (W)  register, α
CMP (W)  register, β
BZ       $??FALSE
```

③ If the control statement is described in uppercase letters and no register name is specified, the following instructions are generated:

```
CMP (W)  α, β
BNZ      $??TRUE
BR       ??FALSE
??TRUE:
```

④ If the control statement is described in uppercase letters and a register name is specified, the following instructions are generated:

```
MOV (W)  register, α
CMP (W)  register, β
BNZ      $??TRUE
BR       ??FALSE
??TRUE:
```

[Application example]

<Input source program>

```
    ; if ( AX != HL )
          CALL    !XXX
    ; else
          CALL    !YYY
    ; endif
```

```
    ; if ( !ABC != #5 )  ( A )
          CALL    !PPP
    ; endif
```

```
    ; IF ( AX != HL )
          CALL    !XXX
    ; ELSE
          CALL    !YYY
    ; ENDIF
```

```
    ; IF ( !ABC != #5 )  ( A )
          CALL    !PPP
    ; ENDIF
```

<Output source program>

```
;  if ( AX != HL )
                        CMPW    AX, HL
                        BZ      $??1
        CALL    !XXX
;  else
                        BR      ??2
                ??1:
        CALL    !YYY
;  endif
                ??2:
```

```
;  if ( !ABC != #5 )  ( A )
                        MOV     A, !ABC
                        CMP     A, #5
                        BZ      $??3
        CALL    !PPP
;  endif
                ??3:
```

```
;  IF ( AX != HL )
                        CMPW    AX, HL
                        BNZ     $??4
                        BR      ??5
                ??4:
        CALL    !XXX
;  ELSE
                        BR      ??6
                ??5:
        CALL    !YYY
;  ENDIF
                ??6:
```

```
;  IF ( !ABC != #5 )  ( A )
                        MOV     A, !ABC
                        CMP     A, #5
                        BNZ     $??7
                        BR      ??8
                ??7:
        CALL    !PPP
;  ENDIF
                ??8:
```

(3) Less Than (<)

[Description format]

α < β [(γ)] γ : register name

[Function]

① If the value of the left term "α"(alpha) is less than the value of the right term "β"(beta), True ("1") is returned; otherwise, False ("0") is returned.

② If γ (register name) is specified, the value of α (alpha) is transferred to the register specified by γ (gamma). If the value of γ (gamma) is less than the value of β (beta), True is returned; otherwise, False is returned.

③ Uppercase and lowercase letters
Instructions to be generated will differ depending on whether the control statement in which the relational operator expression is specified has been described in uppercase or lowercase letters.

Uppercase letters ..... Used when a jump-to-address by the control statement is extremely distant. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ..... Used when a jump-to-address by the control statement is within a range of 128 bytes.

[Explanation]

For α (alpha), β (beta), and γ (gamma) that can be described in this relational operator expression, see Appendix B, "List of Operands".

[Instructions to be generated]

① If the control statement is described in lowercase letters and no register name is specified, the following instructions are generated:

```
        CMP (W)   α, β
        BNC       $??FALSE
```

② If the control statement is described in lowercase letters and a register name is specified, the following instructions are generated:

```
        MOV (W)   register, α
        CMP (W)   register, β
        BNC       $??FALSE
```

③ If the control statement is described in uppercase letters and no register name is specified, the following instructions are generated:

```
        CMP (W)   α, β
        BC        $??TRUE
        BR        ??FALSE
??TRUE:
```

④ If the control statement is described in uppercase letters and a register name is specified, the following instructions are generated:

```
        MOV (W)   register, α
        CMP (W)   register, β
        BC        $??TRUE
        BR        ??FALSE
??TRUE:
```

[Application example]

<Input source program>

```
    if ( A < [HL] )
          CALL   !XXX
    else
          CALL   !YYY
    endif
```

```
    if ( ABCP < HL )  ( AX )
          CALL   !PPP
    endif
```

```
    IF ( A < [HL] )
          CALL   !XXX
    ELSE
          CALL   !YYY
    ENDIF
```

```
    IF ( ABCP < HL )  ( AX )
          CALL   !PPP
    ENDIF
```

<Output source program>

```
;  if ( A < [HL] )
                        CMP     A, [HL]
                        BNC     $??1
        CALL    !XXX
;  else
                        BR      ??2
                ??1:
        CALL    !YYY
;  endif
                ??2:
```

```
;  if ( ABCP < HL )  ( AX )
                        MOVW    AX, ABCP
                        CMPW    AX, HL
                        BNC     $??3
        CALL    !PPP
;  endif
                ??3:
```

```
;  IF ( A < [HL] )
                        CMP     A, [HL]
                        BC      $??4
                        BR      ??5
                ??4:
        CALL    !XXX
;  ELSE
                        BR      ??6
                ??5:
        CALL    !YYY
;  ENDIF
                ??6:
```

```
;  IF ( ABCP < HL )  ( AX )
                        MOVW    AX, ABCP
                        CMPW    AX, HL
                        BC      $??7
                        BR      ??8
                ??7:
        CALL    !PPP
;  ENDIF
                ??8:
```

(4) Greater Than (>)

[Description format]

α > β [(γ)] γ : register name

[Function]

① If the value of the left term "α"(alpha) is greater than the value of the right term "β"(beta), True ("1") is returned; otherwise, False ("0") is returned.

② If γ (register name) is specified, the value of α (alpha) is transferred to the register specified by γ (gamma). If the value of γ (gamma) is greater than the value of β (beta), True is returned; otherwise, False is returned.

③ Uppercase and lowercase letters
Instructions to be generated will differ depending on whether the control statement in which the relational operator expression is specified has been described in uppercase or lowercase letters.

Uppercase letters ..... Used when a jump-to-address by the control statement is extremely distant. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ..... Used when a jump-to-address by the control statement is within a range of 128 bytes.

[Explanation]

For α (alpha), β (beta), and γ (gamma) that can be described in this relational operator expression, see Appendix B, "List of Operands".

[Instructions to be generated]

① If the control statement is described in lowercase letters and no register name is specified, the following instructions are generated:

```
        CMP (W)   α, β
        BZ        $??FALSE
        BC        $??FALSE
```

② If the control statement is described in lowercase letters and a register name is specified, the following instructions are generated:

```
        MOV (W)   register, α
        CMP (W)   register, β
        BZ        $??FALSE
        BC        $??FALSE
```

③ If the control statement is described in uppercase letters and no register name is specified, the following instructions are generated:

```
        CMP (W)   α, β
        BZ        $$+4
        BNC       $??TRUE
        BR        ??FALSE
??TRUE:
```

④ If the control statement is described in uppercase letters and a register name is specified, the following instructions are generated:

```
        MOV (W)   register, α
        CMP (W)   register, β
        BZ        $$+4
        BNC       $??TRUE
        BR        ??FALSE
??TRUE:
```

[Application example]

<Input source program>

```
    if ( B > MEM )
          CALL   !XXX
    else
          CALL   !YYY
    endif
-----------------------------------------
    if ( !ABCP > #0FFFH )  ( AX )
          CALL   !PPP
    endif
-----------------------------------------
    IF ( A > MEM )
          CALL   !XXX
    ELSE
          CALL   !YYY
    ENDIF
-----------------------------------------
    IF ( ABCP > #0FFFH )  ( AX )
          CALL   !PPP
    ENDIF
```

<Output source program>

```
;  if ( B > MEM )
                          CMP     B, MEM
                          BZ      $??1
                          BC      $??1
          CALL   !XXX
;  else
                          BR      ??2
                    ??1:
          CALL   !YYY
;  endif
                    ??2:
-----------------------------------------------------------
```

```
;  if ( !ABCP > #0FFFH )  ( AX )
                          MOVW    AX, !ABCP
                          CMPW    AX, #0FFFH
                          BZ      $??3
                          BC      $??3
          CALL   !PPP
;  endif
                    ??3:
```

```
;  IF ( A > MEM )
                          CMP     A, MEM
                          BZ      $$+4
                          BNC     $??4
                          BR      ??5
                    ??4:
          CALL   !XXX
;  ELSE
                          BR      ??6
                    ??5:
          CALL   !YYY
;  ENDIF
                    ??6:
```

```
;  IF ( ABCP > #0FFFH )  ( AX )
                          MOVW    AX, ABCP
                          CMPW    AX, #0FFFH
                          BZ      $$+4
                          BNC     $??7
                          BR      ??8
                    ??7:
          CALL   !PPP
;  ENDIF
                    ??8:
```

(5) Greater Than or Equal (>=)

[Description format]

α >= β [(γ)] γ : register name

[Function]

① If the value of the left term "α"(alpha) is greater than or equal to the value of the right term "β"(beta), True ("1") is returned; otherwise, False ("0") is returned.

② If γ (register name) is specified, the value of α (alpha) is transferred to the register specified by γ (gamma). If the value of γ (gamma) is greater than or equal to the value of β (beta), True is returned; otherwise, False is returned.

③ Uppercase and lowercase letters
Instructions to be generated will differ depending on whether the control statement in which the relational operator expression is specified has been described in uppercase or lowercase letters.

Uppercase letters ..... Used when a jump-to-address by the control statement is extremely distant. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ..... Used when a jump-to-address by the control statement is within a range of 128 bytes.

[Explanation]

For α (alpha), β (beta), and γ (gamma) that can be described in this relational operator expression, see Appendix B, "List of Operands".

[Instructions to be generated]

① If the control statement is described in lowercase letters and no register name is specified, the following instructions are generated:

```
CMP (W)   α, β
BC        $??FALSE
```

② If the control statement is described in lowercase letters and a register name is specified, the following instructions are generated:

```
MOV (W)   register, α
CMP (W)   register, β
BC        $??FALSE
```

③ If the control statement is described in uppercase letters and no register name is specified, the following instructions are generated:

```
CMP (W)   α, β
BNC       $??TRUE
BR        ??FALSE
??TRUE:
```

④ If the control statement is described in uppercase letters and a register name is specified, the following instructions are generated:

```
MOV (W)   register, α
CMP (W)   register, β
BNC       $??TRUE
BR        ??FALSE
??TRUE:
```

[Application example]

<Input source program>

```
if ( A >= MEM )
        CALL    !XXX
else
        CALL    !YYY
endif
```

```
if ( ABCP >= #0FFFH )  ( AX )
        CALL    !PPP
endif
```

```
IF ( A >= MEM )
        CALL    !XXX
ELSE
        CALL    !YYY
ENDIF
```

```
IF ( ABCP >= #0FFFH )  ( AX )
        CALL    !PPP
ENDIF
```

<Output source program>

```
;  if ( A >= MEM )
                        CMP     A, MEM
                        BC      $??1
        CALL    !XXX
;  else
                        BR      ??2
                ??1:
        CALL    !YYY
;  endif
                ??2:
```

```
;  if ( ABCP >= #0FFFH )  ( AX )
                        MOVW    AX, ABCP
                        CMPW    AX, #0FFFH
                        BC      $??3
        CALL    !PPP
```

```
                ??3:

;  IF ( A >= MEM )
                        CMP     A, MEM
                        BNC     $??4
                        BR      ??5
                ??4:
        CALL    !XXX
;  ELSE
                        BR      ??6
                ??5:
        CALL    !YYY
;  ENDIF
                ??6:
```

```
;  IF ( ABCP >= #0FFFH )  ( AX )
                        MOVW    AX, ABCP
                        CMPW    AX, #0FFFH
                        BNC     $??7
                        BR      ??8
                ??7:
        CALL    !PPP
;  ENDIF
                ??8:
```

(6) Less Than or Equal (<=)

[Description format]

α <= β [(γ)] γ : register name

[Function]

① If the value of the left term "α"(alpha) is less than or equal to the value of the right term "β"(beta), True ("1") is returned; otherwise, False ("0") is returned.

② If γ (register name) is specified, the value of α (alpha) is transferred to the register specified by γ (gamma). If the value of γ (gamma) is less than or equal to the value of β (beta), True is returned; otherwise, False is returned.

③ Uppercase and lowercase letters
Instructions to be generated will differ depending on whether the control statement in which the relational operator expression is specified has been described in uppercase or lowercase letters.

Uppercase letters ..... Used when a jump-to-address by the control statement is extremely distant. In this case, instructions that can be branched to all addresses will be generated.

Lowercase letters ..... Used when a jump-to-address by the control statement is within a range of 128 bytes.

[Explanation]

For α (alpha), β (beta), and γ (gamma) that can be described in this relational operator expression, see Appendix B, "List of Operands".

[Instructions to be generated]

① If the control statement is described in lowercase letters and no register name is specified, the following instructions are generated:

```
        CMP (W)   α, β
        BNZ       $??FALSE
        BNC       $??FALSE
```

② If the control statement is described in lowercase letters and a register name is specified, the following instructions are generated:

```
        MOV (W)   register, α
        CMP (W)   register, β
        BNZ       $??FALSE
        BNC       $??FALSE
```

③ If the control statement is described in uppercase letters and no register name is specified, the following instructions are generated:

```
        CMP (W)   α, β
        BZ        $??TRUE
        BC        $??TRUE
        BR        ??FALSE
??TRUE:
```

④ If the control statement is described in uppercase letters and a register name is specified, the following instructions are generated:

```
        MOV (W)   register, α
        CMP (W)   register, β
        BZ        $??TRUE
        BC        $??TRUE
        BR        ??FALSE
??TRUE:
```

[Application example]

<Input source program>

```
    if ( A <= MEM )
          CALL   !XXX
    else
          CALL   !YYY
    endif
-----------------------------------------
    if ( &[DE] <= #0FFFH )  ( AX )
          CALL   !PPP
    endif
-----------------------------------------
    IF ( A <= MEM )
          CALL   !XXX
    ELSE
          CALL   !YYY
    ENDIF
-----------------------------------------
    IF ( &[DE] <= #0FFFH )  ( AX )
          CALL   !PPP
    ENDIF
```

<Output source program>

```
;  if ( A <= MEM )
                          CMP     A,MEM
                          BNZ     $??1
                          BNC     $??1
          CALL   !XXX
;  else
                          BR      ??2
                    ??1:
          CALL   !YYY
;  endif
                    ??2:
-----------------------------------------------------------
```

```
; if ( &[DE] <= #0FFFH )  ( AX )
                        MOVW    AX,&[DE]
                        CMPW    AX,#0FFFH
                        BNZ     $??3
                        BNC     $??3
        CALL    !PPP
; endif
                    ??3:
```

```
; IF ( A <= MEM )
                        CMP     A,MEM
                        BZ      $??4
                        BC      $??4
                        BR      ??5
                    ??4:
        CALL    !XXX
; ELSE
                        BR      ??6
                    ??5:
        CALL    !YYY
; ENDIF
                    ??6:
```

```
; IF ( &[DE] <= #0FFFH )  ( AX )
                        MOVW    AX,&[DE]
                        CMPW    AX,#0FFFH
                        BZ      $??7
                        BC      $??7
                        BR      ??8
                    ??7:
        CALL    !PPP
; ENDIF
                    ??8:
```

(7) bit symbol

[Description format]

```
if_bit (bit symbol)
elseif_bit (bit symbol)
while_bit (bit symbol)
until_bit (bit symbol)
```

[Function]

If the value of a specified bit symbol is "1", True is returned; if it is "0", False is returned.
In the following control statements, a bit symbol can be described as a condition expression.

if_bit
elseif_bit
while_bit
until_bit

[Explanation]

For the bit symbols that can be described in this bit condition expression, see Appendix B, "List of Operands". Symbols assigned to these bit symbols with EQU directive may also be described.

Example:
```
if_bit ( AAA.3 )
BBB  EQU trgf.1
if_bit ( BBB )
```

[Instructions to be generated]

Control statement	Bit symbol	Instruction to be generated
When described in lowercase letters	CY	BNC $??FALSE
	Z	BNZ $??FALSE
	Others	BF bit symbol,$??FALSE
When described in uppercase letters	CY	BC $??TRUE BR ??FALSE ??TRUE:
	Z	BZ $??TRUE BR ??FALSE ??TRUE:
	Others	BT bit symbol,$??TRUE BR ??FALSE ??TRUE:

[Application example]

<Input source program>

```
    if_bit ( TRFG.0 )
           CALL  !XXX
    else
           CALL  !YYY
    endif
-----------------------------------
    if_bit ( CY )
           CALL  !XXX
    else
           CALL  !YYY
    endif
-----------------------------------
    if_bit ( Z )
           CALL  !XXX
    else
           CALL  !YYY
    endif
-----------------------------------
    IF_BIT ( TRFG.0 )
           CALL  !XXX
    ELSE
           CALL  !YYY
    ENDIF
-----------------------------------
    IF_BIT ( CY )
           CALL  !XXX
    ELSE
           CALL  !YYY
    ENDIF
-----------------------------------
    IF_BIT ( Z )
           CALL  !XXX
    ELSE
           CALL  !YYY
    ENDIF
```

<Output source program>

```
;  if_bit ( TRFG.0 )
                        BF      TRFG.0, $??1
        CALL  !XXX
;  else
                        BR      ??2
                  ??1:
        CALL  !YYY
;  endif
                  ??2:
```

```
;  if_bit ( CY )
                        BNC     $??3
        CALL  !XXX
;  else
                        BR      ??4
                  ??3:
        CALL  !YYY
;  endif
                  ??4:
```

```
;  if_bit ( Z )
                        BNZ     $??5
        CALL  !XXX
;  else
                        BR      ??6
                  ??5:
        CALL  !YYY
;  endif
                  ??6:
```

```
;  IF_BIT ( TRFG.0 )
                        BT      TRFG.0, $??7
                        BR      ??8
                  ??7:
        CALL  !XXX
;  ELSE
                        BR      ??9
                  ??8:
        CALL  !YYY
;  ENDIF
                  ??9:
```

```
;  IF_BIT ( CY )
                        BC      $??10
                        BR      ??11
                    ??10:
           CALL  !XXX
;  ELSE
                        BR      ??12
                    ??11:
           CALL  !YYY
;  ENDIF
                    ??12:
```

```
;  IF_BIT ( Z )
                        BZ      $??13
                        BR      ??14
                    ??13:
           CALL  !XXX
;  ELSE
                        BR      ??15
                    ??14:
           CALL  !YYY
;  ENDIF
                    ??15:
```

(8) **!bit symbol**

[Description format]

```
if_bit (!bit symbol)
elseif_bit (!bit symbol)
while_bit (!bit symbol)
until_bit (!bit symbol)
```

[Function]

If the value of a specified bit symbol is "0", True is returned; if it is "1", False is returned.
In the following control statements, a bit symbol can be described as a condition expression.

if_bit
elseif_bit
while_bit
until_bit

[Explanation]

For the bit symbols that can be described in this bit condition expression, see Appendix B, "List of Operands". Symbols assigned to these bit symbols with EQU directive may also be described.

Example:
```
if_bit ( !AAA.3 )
BBB  EQU trgf.1
if_bit ( !BBB )
```

[Instructions to be generated]

Control statement	Bit symbol	Instruction to be generated
When described in lowercase letters	CY	BC $??FALSE
	Z	BZ $??FALSE
	Others	BT bit symbol,$??FALSE
When described in uppercase letters	CY	BNC $??TRUE BR ??FALSE ??TRUE:
	Z	BNZ $??TRUE BR ??FALSE ??TRUE:
	Others	BF bit symbol,$??TRUE BR ??FALSE ??TRUE:

[Application example]

<Input source program>

```
    if_bit ( !TRFG.0 )
            CALL  !XXX
    else
            CALL  !YYY
    endif
-----------------------------------
    if_bit ( !CY )
            CALL  !XXX
    else
            CALL  !YYY
    endif
-----------------------------------
    if_bit ( !Z )
            CALL  !XXX
    else
            CALL  !YYY
    endif
-----------------------------------
    IF_BIT ( !TRFG.0 )
            CALL  !XXX
    ELSE
            CALL  !YYY
    ENDIF
-----------------------------------
    IF_BIT ( !CY )
            CALL  !XXX
    ELSE
            CALL  !YYY
    ENDIF
-----------------------------------
    IF_BIT ( !Z )
            CALL  !XXX
    ELSE
            CALL  !YYY
    ENDIF
```

<Output source program>

```
; if_bit ( !TRFG.0 )
                        BT      TRFG.0, $??1
        CALL  !XXX
; else
                        BR      ??2
                  ??1:
        CALL  !YYY
; endif
                  ??2:
-------------------------------------------------------------
; if_bit ( !CY )
                        BC      $??3
        CALL  !XXX
; else
                        BR      ??4
                  ??3:
        CALL  !YYY
; endif
                  ??4:
-------------------------------------------------------------
; if_bit ( !Z )
                        BZ      $??5
        CALL  !XXX
; else
                        BR      ??6
                  ??5:
        CALL  !YYY
; endif
                  ??6:
-------------------------------------------------------------
; IF_BIT ( !TRFG.0 )
                        BF      TRFG.0, $??7
                        BR      ??8
                  ??7:
        CALL  !XXX
; ELSE
                        BR      ??9
                  ??8:
        CALL  !YYY
; ENDIF
                  ??9:
-------------------------------------------------------------
```

```
; IF_BIT ( !CY )
                        BNC     $??10
                        BR      ??11
                  ??10:
        CALL  !XXX
; ELSE
                        BR      ??12
                  ??11:
        CALL  !YYY
; ENDIF
                  ??12:
```

```
; IF_BIT ( !Z )
                        BNZ     $??13
                        BR      ??14
                  ??13:
        CALL  !XXX
; ELSE
                        BR      ??15
                  ??14:
        CALL  !YYY
; ENDIF
                  ??15:
```

(9) AND (&&)

[Description format]

condition expression 1 && condition expression 2

[Function]

Performs an AND operation between condition expression 1 and condition expression 2. If both condition expression 1 and condition expression 2 are true, True ("1") is returned; otherwise, False ("0") is returned.
This expression is used to describe processing when two conditions are met at the same time.

[Explanation]

① A relational operation expression or bit condition expression must be described as condition expression 1 and condition expression 2.

② If a register name is specified in a control statement, the specified register is used for True/False judgment on both condition expression 1 and condition expression 2.

[Instructions to be generated]

Example: α1 == β1 && α2 != β2

① When the control statement is described in lowercase letters

```
CMP (W)   α1, β1   ┐ Condition expression 1
BNZ       $??FALSE ┘
CMP (W)   α2, β2   ┐ Condition expression 2
BNZ       $??FALSE ┘
```

② When the control statement is described in uppercase letters

```
          CMP (W)   α1, β          ┐
          BZ        $??TRUE1       ├  Condition expression 1
          BR        ??FALSE        ┘
??TRUE1:
          CMP (W)   α2, β2         ┐
          BZ        $??TRUE2       ├  Condition expression 2
          BR        ??FALSE        ┘
??TRUE2:
```

[Application example]

<Input source program>

```
if ( A == #0 && B != #0 )
        CALL  !XXX
 else
        CALL  !YYY
 endif
```

```
IF ( A == #0 && B != #0 )
        CALL  !XXX
 ELSE
        CALL  !YYY
 ENDIF
```

<Output source program>

```
;  if ( A == #0 && B != #0 )
                                        CMP     A, #0
                                        BNZ     $??1
                                        CMP     B, #0
                                        BZ      ??1
          CALL  !XXX
;    else
                                        BR      ??2
                                ??1:
          CALL  !YYY
;    endif
                                ??2:
```

```
;  IF ( A == #0 && B != #0 )
                                        CMP     A, #0
                                        BZ      $??1
                                        BR      ??2
                                ??1:
                                        CMP     B, #0
                                        BNZ     $??3
                                        BR      ??2
                                ??3:
          CALL  !XXX
;    ELSE
                                        BR      ??4
                                ??2:
          CALL  !YYY
;    ENDIF
                                ??4:
```

(10) OR (| |)

[Description format]

condition expression 1		condition expression 2

[Function]

Performs an OR operation between condition expression 1 and condition expression 2. If either condition expression 1 or condition expression 2 is true, True ("1") is returned; if both are false, False ("0") is returned.
This expression is used to describe processing when either of two conditions is met.

[Explanation]

① A relational operation expression or bit condition expression must be described as condition expression 1 and condition expression 2.

② If a register name is specified in a control statement, the specified register is used for True/False judgment on both condition expression 1 and condition expression 2.

[Instructions to be generated]

Example: $\alpha 1$ == $\beta 1$ | | $\alpha 2$!= $\beta 2$

① When the control statement is described in lowercase letters

```
        CMP (W)   α1, β1   ┐
        BZ        $??TRUE  ┘  Condition expression 1
        CMP (W)   α2, β2   ┐
        BZ        $??FALSE ┘  Condition expression 2
??TRUE:
```

② When the control statement is described in uppercase letters

```
        CMP (W)   α1, β1  ┐
        BZ        $??TRUE ┘   Condition expression 1

        CMP (W)   α2, β2  ┐
        BNZ       $??TRUE ├   Condition expression 2
        BR        ??FALSE ┘
??TRUE:
```

[Application example]

<Input source program>

```
if ( A == #0 || B != #0 )
        CALL  !XXX
else
        CALL  !YYY
endif
```

```
IF ( A == #0 || B != #0 )
        CALL  !XXX
ELSE
        CALL  !YYY
ENDIF
```

<Output source program>

```
;  if ( A == #0 || B != #0 )
                                        CMP     A, #0
                                        BZ      $??1
                                        CMP     B, #0
                                        BZ      $??2
                                ??1:
           CALL  !XXX
;    else
                                        BR      ??3
                                ??2:
           CALL  !YYY
;    endif
                                ??3:
```

```
;  IF ( A == #0 || B != #0 )
                                        CMP     A, #0
                                        BZ      $??1
                                        CMP     B, #0
                                        BNZ     $??1
                                        BR      ??2
                                ??1:
           CALL  !XXX
;    ELSE
                                        BR      ??3
                                ??2:
           CALL  !YYY
;    ENDIF
                                ??3:
```

CHAPTER 4. EXPRESSION STATEMENTS

Expression statements are used to execute substitutions and arithmetic and other operations. These statements are divided into the following three types:

- o Substitution statement .... Substitutes the first (left) term of an expression with its second (right) term.
- o Counter statement ..... Increments or decrements the value of the term of an expression by 1.
- o Exchange statement ..... Exchanges the value of the first (left) term of an expression for that of its second (right) term.

Substitution statement	Description format	Function
(1) Substitute	$\alpha = \beta$	$\alpha \leftarrow \beta$
(2) Substitute with Register name specification	$\alpha = \beta \quad (\gamma)$	$\gamma \leftarrow \beta \quad \alpha \leftarrow \gamma$
(3) Add & Substitute	$\alpha += \beta$	$\alpha \leftarrow \alpha + \beta$
(4) Subtract & Substitute	$\alpha -= \beta$	$\alpha \leftarrow \alpha - \beta$
(5) Multiply & Substitute	$\alpha *= \beta$	$\alpha \leftarrow \alpha \times \beta$
(6) Divide & Substitute	$\alpha /= \beta$	$\alpha \leftarrow \alpha \div \beta$
(7) AND & Substitute	$\alpha \&= \beta$	$\alpha \leftarrow \alpha \cap \beta$
(8) OR & Substitute	$\alpha \mid = \beta$	$\alpha \leftarrow \alpha \cup \beta$
(9) XOR & Substitute	$\alpha \hat{} = \beta$	$\alpha \leftarrow \alpha \hat{} \beta$
(10) Shift Right & Substitute	$\alpha >>= \beta$	$(CY \leftarrow \alpha_0 \ \alpha_{n-1} \leftarrow \alpha_n \ \alpha_{max} \leftarrow 0) \times \beta$ times
(11) Shift Left & Substitute	$\alpha <<= \beta$	$(CY \leftarrow \alpha_{max} \ \alpha_{n+1} \leftarrow \alpha_n \ \alpha_0 \leftarrow 0) \times \beta$ times

Counter statement	Description format	Function
(12) Increment	$\alpha++$	$\alpha \leftarrow \alpha + 1$
(13) Decrement	$\alpha--$	$\alpha \leftarrow \alpha - 1$

Exchange statement	Description format	Function
(14) Exchange	$\alpha <-> \beta$	$\alpha \leftarrow \alpha <-> \beta$

The description format, function, and application example(s) of each expression statement are explained below.

(1) Substitute (=)

[Description format]

$\alpha = \beta$

[Function]

Substitutes the left term "α"(alpha) of an expression with its right term "β"(beta).

[Explanation]

For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) or right term "β"(beta) of the expression is CY (Carry flag), the following instruction is generated:

MOV1 α, β

② If the left term "α" or right term "β" is a word symbol in cases other than ① above, the following instruction is generated:

MOVW α, β

③ In cases other than ① and ② above, the following instruction is generated:

MOV α, β

[Application example]

<Input source file>

```
CY = P0.1
AX = CR00
A = #0FEH
```

<Output source file>

```
;CY = P0.1
                    MOV1    CY,P0.1

;AX = CR00
                    MOVW    AX,CR00

;A = #0FEH
                    MOV     A,#0FEH
```

(2) Substitute with Register name specification (=)

[Description format]

$\alpha = \beta \triangle (\gamma)$ γ : register name or CY

[Function]

Substitutes the contents of "γ"(gamma) with the right term "β"(beta) of an expression and then substitute the left term "α" (alpha) of the expression with the contents of γ (gamma).

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② Note that the contents of the register specified by γ (gamma) may be changed by the contents of β (beta).

③ A blank (space) character must be inserted between the right term "β" and (γ) (gamma).

[Instructions to be generated]

① If γ (gamma) is CY (Carry flag), the following instructions are generated:

```
MOV1    CY, β
MOV1    α, CY
```

② If γ (gamma) is a register pair, the following instructions are generated:

```
MOVW    γ, β
MOVW    α, γ
```

③ In cases other than ① and ② above, the following instruction is generated:

```
MOV     specified register, β
MOV     α , specified register
```

[Application example]

<Input source file>

```
A.0 = TFLG.0  (CY)
BC = TM0  (AX)
!ABC = #0FCH  (A)
```

<Output source file>

```
;A.0 = TFLG.0  (CY)
                        MOV1    CY, TFLG.0
                        MOV1    A.0, CY

;BC = TM0  (AX)
                        MOVW    AX, TM0
                        MOVW    BC, AX

;!ABC = #0FCH  (A)
                        MOV     A, #0FCH
                        MOV     !ABC, A
```

(3) Add & Substitute (+=)

[Description format]

α += β

[Function]

Performs addition between the two terms of an expression ($\alpha + \beta$) and substitutes the left term " α " (alpha) of the expression with the result of the addition.

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② If addition with carry is to be performed, describe the assembly language as is.

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term " α " (alpha) of the expression is a word symbol, the following instruction is generated. (With the ST78K2, only AX or RP0 can be described as a word symbol.)

```
ADDW    α, β
```

② In cases other than ① above, the following instruction is generated:

```
ADD     α, β
```

[Application example]

<Input source file>

```
AX += #0C000H
A += #0C0H
```

<Output source file>

```
;AX += #0C000H
                    ADDW    AX,#0C000H

;A += #0C0H
                    ADD     A,#0C0H
```

(4) Subtract & Substitute (-=)

[Description format]

α -= β

[Function]

Performs subtraction between the two terms of an expression ($\alpha - \beta$) and substitutes the left term "α" (alpha) of the expression with the result of the subtraction.

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② If subtraction with carry is to be performed, describe the assembly language as is.

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) of the expression is a word symbol, the following instruction is generated. (With the ST78K2, only AX or RP0 can be described as a word symbol.)

SUBW α, β

② In cases other than ① above, the following instruction is generated:

SUB α, β

[Application example]

<Input source file>

```
AX -= #0C000H
A -= #0C0H
```

<Output source file>

```
;AX -= #0C000H
                    SUBW    AX,#0C000H

;A -= #0C0H
                    SUB     A,#0C0H
```

(5) Multiply & Substitute (*=)

[Description format]

α *= β

[Function]

Performs multiplication between the two terms of an expression ($\alpha \times \beta$) and substitutes the left term "α"(alpha) of the expression with the result of the multiplication.

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② If the right term "β"(beta) of the expression is not a register pair, the contents of the A register are multiplied by the value of the right term "β"(beta) of the expression and then the contents of the AX register are substituted with the result of the multiplication.

③ If the right term "β"(beta) of the expression is a register pair, the contents of the AX register are multiplied by the value of the right term "β"(beta) and then the contents of the AX register are substituted with the high-order 16 bits of the result of the multiplication and the contents of the right term "β"(beta) are substituted with the low-order 16 bits of the result.

④ Either AX or RP0 must be described as the left term "α" (alpha) of the expression. If the right term "β"(beta) is a register pair, the register pair may be described following AX or RP0. For example, AX, BC <- AX x BC may be described as either of the following two:

```
AX *= BC
AXBC *= BC
```

⑤ With a signed Multiply instruction, describe the assembly language as is.

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the right term "β"(beta) of the expression is a register pair, the following instruction is generated:

```
MULUW    β
```

② In cases other than ① above, the following instruction is generated:

```
MULU     β
```

[Application example]

<Input source file>

```
AX *= B
AXDE *= DE
AX *= DE
```

<Output source file>

```
;AX *= B
                MULU    B
;AXDE *= DE
                MULUW   DE
;AX *= DE
                MULUW   DE
```

(6) Divide & Substitute (/=)

[Description format]

α /= β

[Function]

Performs division between the two terms of an expression ($\alpha \div \beta$) and substitutes the left term "α"(alpha) of the expression with the result of the division.

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② If the right term "β"(beta) of the expression is not a register pair, the contents of the A register are divided by the value of the right term "β"(beta) of the expression and then the contents of the AX register are substituted with the quotient in the result of the division and the contents of the right term "β"(beta) are substituted with the remainder of the result.

③ If the right term "β"(beta) of the expression is a register pair, the contents of the AXDE registers are divided by the value of the right term "β"(beta) and then the contents of the AXDE registers are substituted with the quotient in the result of the division and the contents of the right term "β"(beta) are substituted with the remainder of the result.

④ Either AX or RP0 must be described as the left term "α" (alpha) of the expression.

⑤ With a signed Divide instruction, describe the assembly language as is.

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the right term "β"(beta) of the expression is a register pair, the following instruction is generated:

```
DIVUX    β
```

② In cases other than ① above, the following instruction is generated:

```
DIVUW    β
```

[Application example]

<Input source file>

```
AXDE /= BC
AX /= E
```

<Output source file>

```
;AXDE /= BC
                DIVUX    BC
;AX /= E
                DIVUW    E
```

(7) AND & Substitute (&=)

[Description format]

α &= β

[Function]

Performs an AND operation between the two terms of an expression ($\alpha \cap \beta$) on a bit-to-bit basis and substitutes the left term "α"(alpha) of the expression with the result of the AND operation.

[Explanation]

For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) of the expression is CY (Carry flag), the following instruction is generated:

```
AND1    CY, β
```

② In cases other than ① above, the following instruction is generated:

```
AND     α, β
```

[Application example]

<Input source file>

```
CY &= P0.1
A &= #0FFH
```

<Output source file>

```
;CY &= P0.1
                AND1    CY,P0.1
;A &= #0FFH
                AND     A,#0FFH
```

(8) OR & Substitute (| =)

[Description format]

α | = β

[Function]

Performs an OR operation between the two terms of an expression ($\alpha \cup \beta$) on a bit-to-bit basis and substitutes the left term "α"(alpha) of the expression with the result of the OR operation.

[Explanation]

For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) of the expression is CY (Carry flag), the following instruction is generated:

```
OR1     CY, β
```

② In cases other than ① above, the following instruction is generated:

```
OR      α, β
```

[Application example]

<Input source file>

```
CY |= P0.1
A |= #0FFH
```

<Output source file>

```
;CY |= P0.1
                    OR1     CY, P0.1
;A |= #0FFH
                    OR      AX, #0FFH
```

(9) XOR & Substitute (^ =)

[Description format]

α ^= β

[Function]

Performs an XOR operation between the two terms of an expression (α ^ β) on a bit-to-bit basis and substitutes the left term "α"(alpha) of the expression with the result of the XOR operation.

[Explanation]

For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The left and right terms of the expression are evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) of the expression is CY (Carry flag), the following instruction is generated:

```
XOR1    CY, β
```

② In cases other than ① above, the following instruction is generated:

```
XOR     α, β
```

[Application example]

<Input source file>

```
CY ^= PSW.1
A ^= #0FFH
```

<Output source file>

```
;CY ^= PSW.1
                XOR1    CY,PSW.1
;A ^= #0FFH
                XOR     A,#0FFH
```

(10) Shift Right & Substitute (>>=)

[Description format]

α >>= β

[Function]

Shifts to the right the value of the left term "α"(alpha) of an expression by the number of bits specified by the right term "β"(beta) of the expression and substitutes the contents of "α"(alpha) with the result of the Shift Right operation.

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② As the right term "β"(beta) of the expression, describe the number of bits to be shifted without adding "#" to it.

[Instructions to be generated]

The left term of the expression is evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) of the expression is a word symbol, the following instruction is generated:

```
SHRW    α, β
```

② In cases other than ① above, the following instruction is generated:

```
SHR     α, β
```

[Application example]

<Input source file>

```
AX >>= 7
A >>= 4
```

<Output source file>

```
;AX >>= 7
                SHRW    AX,7
;A >>= 4
                SHR     A,4
```

(11) Shift Left & Substitute (<<=)

[Description format]

α <<= β

[Function]

Shifts to the left the value of the left term "α"(alpha) of an expression by the number of bits specified by the right term "β" (beta) of the expression and substitutes the contents of "α"(alpha) with the result of the Shift Left operation.

[Explanation]

① For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

② As the right term "β"(beta) of the expression, describe the number of bits to be shifted without adding "#" to it.

[Instructions to be generated]

The left term of the expression is evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the left term "α"(alpha) of the expression is a word symbol, the following instruction is generated:

SHLW α, β

② In cases other than ① above, the following instruction is generated:

SHL α, β

[Application example]

<Input source file>

```
AX <<= 7
A <<= 4
```

<Output source file>

```
;AX <<= 7
                SHLW    AX,7
;A <<= 4
                SHL     A,4
```

(12) Increment (++)

[Description format]

α ++

[Function]

Increments the contents (value) of the term "α"(alpha) of an expression by 1.

[Explanation]

For α (alpha) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The term of the expression is evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the term "α"(alpha) of the expression is a word symbol, the following instruction is generated:

INCW α

② In cases other than ① above, the following instruction is generated:

INC α

[Application example]

<Input source file>

```
HL++
B++
CNT++
```

<Output source file>

```
;HL++
                INCW    HL
;B++
                INC     B
;CNT++
                INC     CNT
```

(13) Decrement (--)

[Description format]

α --

[Function]

Decrements the contents (value) of the term "α"(alpha) of an expression by 1.

[Explanation]

For α (alpha) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The term of the expression is evaluated in the following order and the applicable instruction is generated according to the evaluation result.

① If the term "α"(alpha) of the expression is a word symbol, the following instruction is generated:

```
DECW    α
```

② In cases other than ① above, the following instruction is generated:

```
DEC     α
```

[Application example]

<Input source file>

```
HL--
B--
CNT--
```

<Output source file>

```
;HL--
                    DECW    HL
;B--
                    DEC     B
;CNT--
                    DEC     CNT
```

(14) Exchange (<->)

[Description format]

```
α <-> β
```

[Function]

Exchanges the contents (value) of the left term "α"(alpha) of an expression for the contents of its right term "β" (beta).

[Explanation]

For α (alpha) and β (beta) that can be described in this expression statement, see Appendix B, "List of Operands".

[Instructions to be generated]

The following instruction is generated:

```
XCH α, β
```

[Application example]

<Input source file>

```
A <-> PM0
```

<Output source file>

```
;A <-> PM0
        XCH     A,PM0
```

CHAPTER 5. DIRECTIVES

5.1 Overview of Directives

Structured assembler directives (pseudoinstructions) must be described in a source program.
These assembler directives provide various directions required when the structured assembler preprocessor performs a series of operations.
By using these directives, a source program can be written with ease. However, the directives described in the source program will not be listed in the output file.

5.2 Function of Each Directive

Table 5-1 lists the directives that can be used with this assembler package.

Table 5-1. List of Directives

Type of directive	Directive
Identifier defining directive	#define
Conditional processing directives	#ifdef : #else : #endif
Include directive	#include
CALLT substituting directives	#defcallt : #endcallt

The description format, function, and application example(s) of each directive are explained below.

(1) Identifier defining directive (#define)

[Description format]

```
#define  identifier   character string
```

[Function]

Replaces the identifier defined in the first operand field with the character string specified in the second operand field.

[Explanation]

① The **#define** line must begin with character "#", excluding a blank or HT character.

② An "identifier" may consist of alphanumeric characters and must always begin with an alphabetic character, but only the first eight characters of the identifier will be accepted as valid.

③ A "character string" must consist of characters as described in 2.2.1, Character set. Neither a blank nor a quotation mark can be described in a character string. If used, the structured assembler will continue processing by ignoring the blank or quotation mark.

④ **#define** is effective for describing numerical values with a defined, easy-to-read identifier.

⑤ No reserved word can be used as an identifier.

[Application example]

<Input source program>

```
#define  TRUE   #1
  ............
A  =  #0
if ( A == TRUE )
    AX = #0C5H
endif
```

<Output source program>

```
#define  TRUE   #1
  ............
;    A  =  #0
                          MOV     A,#0
;    if ( A == TRUE )
                          CMP     A,#1
                          BNZ     $??1
;      AX = #0C5H
                          MOV     AX,#0C5H
;    endif
                    ??1:
```

(2) Conditional processing directives (#ifdef/#else/#endif)

[Description format]

```
#ifdef  identifier
   text 1
#else
   text 2
#endif
```

[Function]

Executes conditional processing as follows:

① If the value of "identifier" is 0 (=undefined), text 1 will be skipped and text 2 will be processed.

② If the value of "identifier" is other than 0, text 1 will be processed and text 2 will be skipped.

[Explanation]

① The **#ifdef** line must begin with character "#" excluding a blank or HT character.

② An "identifier" may consist of alphanumeric characters and must always begin with an alphabetic character, but only the first eight characters of the identifier will be accepted as valid.

③ The specified "identifier" must be the one which has been predefined by a **#define** directive or which has been defined by the option "D" in the start-up command line of the structured assembler.

④ **#ifdef** directives may be nested up to eight levels.

⑤ **#else** directive may be omitted from description.

⑥ The **#ifdef** directive must be properly paired with a **#endif** statement, or an error will result.

[Application example]

<TEST.S>

```
#ifdef   D78K2
         text 1
#else
         text 2
#endif
```

<How to start up the structured assembler>

```
A>ST78K2 TEST.S  -DD=78K2=0
```

In this case, text 1 will be processed by this structured assembler and text 2 will be skipped (namely, it will not be processed by the structured assembler).

(3) Include directive (#include)

[Description format]

```
#include  "filename"
```

[Function]

Replaces this **#include** line with the contents of the file specified by the operand "filename" and processes the line as the source program of this preprocessor.

[Explanation]

① The **#include** line must begin with character "#" excluding a blank or HT (Horizontal Tab) character.

② **#include** may be described anywhere in the source program.

③ No **#include** directive can be described in an include file. In other words, nesting of **#include** directives is not allowed.

④ The input source filename specified in the start-up command line of the structured assembler cannot be specified as the operand "filename" of this directive.

⑤ A drive name and a directive name may be described at the beginning of the operand "filename". If these names are omitted, the structured assembler will process this directive by assuming that the include file exists in the current directory on the current drive.

⑥ The drive name and directory name of an include file may be specified in the start-up command line of the structured assembler by using the option "I".

[Application example]

<Input source file>

```
#include   "FILE"

        A=SIZE1
        B=SIZE2
```

<FILE>

```
#define   SIZE1 #08H
#define   SIZE2 #0AH
```

<Output source file>

```
;A=SIZE1
                        MOV     A,#08H
:B=SIZE2
                        MOV     B,#0AH
```

In this case, if "-IB: ¥SRC¥" (with MS-DOS) has been specified at the time of starting up the structured assembler, #include "FILE" directive will cause the file "B:¥SRC¥FILE" to be read by the preprocessor for processing.

Note: With PC-DOS, \(backslash) is used in place of ¥ indicating a directory.

(4) CALLT substituting directives (#defcallt/#endcallt)

[Description format]

```
#defcallt  label of CALLT table
    CALL   label
      :
#endcallt
```

[Function]

Substitutes CALL instruction(s) to registered label(s) with CALLT instruction(s) and outputs them to a secondary source file.

[Explanation]

① In a source program, describe CALL instructions without using CALLT and define the labels registered in the CALLT table with this directive. By so doing, all the CALL instructions to defined labels will also be converted into CALLT instructions.

② This directive can be described a maximum of 32 times in a source program.

③ The **#defcallt** directive must be properly paired with a **#endcallt** statement, or an error will result.

[Application example]

<Input source file>

```
#defcallt   @TAB1
     CALL   !ABC
#endcallt
     CALL   !ABC
     CALL   !BCD
```

<Output source file>

```
:    CALL   !ABC
                       CALLT     [@TAB1]
     CALL   !BCD
```

CHAPTER 6. PRODUCT OVERVIEW

6.1 Contents of Product

This product offers the files listed in Table 6-1 below. Namely, the following files are included in the RA78K series assembler package:

Table 6-1. Files Offered

Filename	Type of file
ST78Kn.EXE	Command file
ST78Kn.OM1	Overlay file
TEST1.S TEST2.S TESTINC.S	Sample program files
ST.BAT	Batch file

n: Represents 0 to 6 as follows:

0 ... 78K0
1 ... 78K1
2 ... 78K2
3 ... 78K3
6 ... 78K6

Command file ..... A file to be first read into memory when each program is activated.

Overlay file ..... A file to be read into memory only when required during the execution of each program.

Sample program files ... Files used to confirm the proper operation of the structured assembler.

Batch file ..... A file to automatically start up the assembler if no error occurs after the structured assembler has been run.

When using the version 2.11 of MS-DOS, the file "ST78Kn.OM1" must be set on the current drive.

6.2 System Configuration

Host computer		OS	Memory size
Series	Applicable model		
PC-9800	PC-9801 E, PC-9801 F1/F2/F3* PC-9801 M2/M3, PC-9801 VF2* PC-9801 VM0/VM2/VM4/VM21 PC-9801 VX0/VX2/VX4 PC-9801 XLmodel/2/4**	MS-DOS Versions 2.11, 3.1 ***	128K bytes or more
IBM PC	IBM PC IBM PC/XT	PC-DOS Versions 3.1, 3.3	

* When using any of these host computers, an extension drive for 8" 2D or 5" 2HD floppy disks must be prepared.

** Use these host computers in the Normal mode.

*** This program (structured assembler) operates on the MS-DOS that NEC offers for the PC-9800 series personal computers. NEC assumes no liability for improper operation of this program on any other commercially available MS-DOS.

CHAPTER 7. OPERATING PROCEDURES

7.1 Types of I/O Files

The structured assembler has the input and output files listed in Table 7-1 below.

Table 7-1. Input/Output Files of Structured Assembler

	Type of file	Default file type
Input files	Source module file A source module file written in the structured assembly language.	
	Include file A file containing a source program that can be expanded into the input source module file.	
	Parameter file (see Note below) A file which contains the options of the structured assembler so that they may be specified from this file.	.PST
Output files	Secondary source module file A source module file written in the assembly language.	.ASM
	Error list file A file containing information on errors in processing by the structured assembler.	.EST

Note: See 7.3.1 (2) "Start-up with parameter file".

7.2 Option Functions

The structured assembler is provided with option functions to facilitate its use. These option functions can be specified in the start-up command line of the structured assembler or by using a parameter file. See Section 7.4, "Structured Assembler Options" for the function of each option.

7.3 How to Start Up Structured Assembler

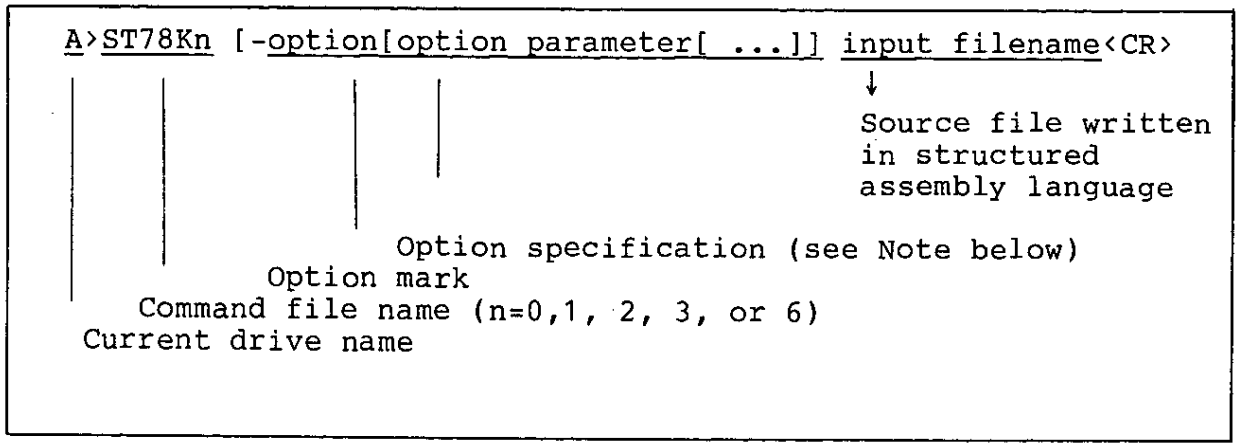
7.3.1 Starting up the structured assembler

The structured assembler can be started up in either of the following two ways:

- o Start-up with command line
- o Start-up with parameter file

(1) Start-up with command line

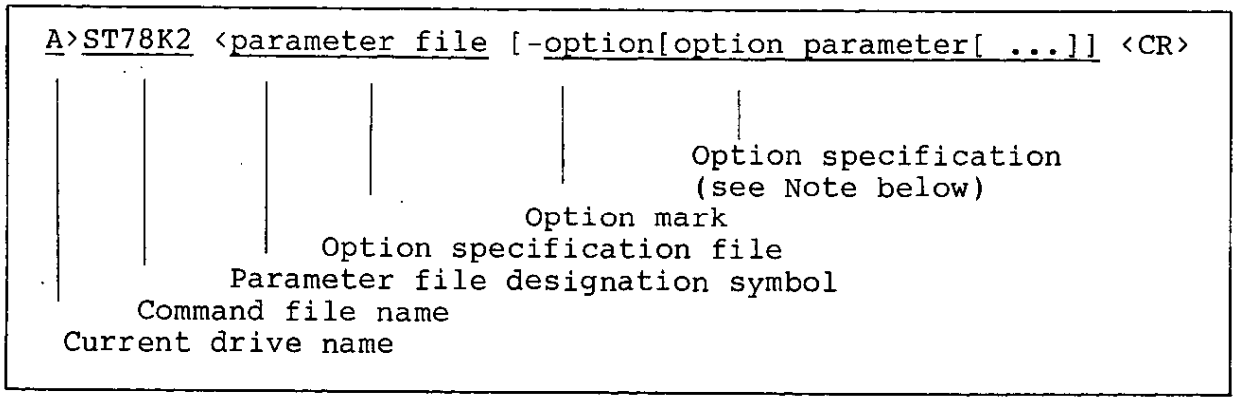
Start up the structured assembler by entering the following command line:



Note: When specifying two or more options, delimit each option with a blank (space).

(2) Start-up with parameter file

When starting up the structured assembler, repeating the same specifications over and over again is troublesome. In such a case, use a parameter file to start up the structured assembler.



Create a parameter file with the editor.
Describe the source filename and the options of the structured assembler in the parameter file.
Any of the options specified in the parameter file may be changed by specifying it in the start-up command line of the structured assembler.

Example 1: To create parameter file "ST.JOB" with editor

o Contents of ST.JOB

```
TEST1.S -WT4,5,6
```

o To start up the structured assembler by specifying parameter file "ST.JOB"

```
A>ST78K2 <ST.JOB
Structured assembler preprocessor for RA78K2 VX.X  [dd Mmm yy]
   Copyright (C) NEC Corporation 1988  US9876543210

conversion complete, 0 error(s) found.

A>
```

Example 2: To change an option specified in parameter file or add an option to parameter file by specifying it in start-up command line

```
A>ST78K2 <ST.JOB  -OB:

A>
```

7.3.2 Execution start and complete messages

(1) Execution start message

On start-up of the structured assembler, the following execution start message is displayed on the console:

```
Structured assembler preprocessor for RA78Kn VX.X  [dd Mmm yy]
   Copyright (C) NEC Corporation 1988  US9876543210
```

(2) Process display message

```
start...
```

"." is displayed on processing of every 100 lines.

(3) Execution complete message

o If no error is detected, the structured assembler outputs the following message on the console on completion of its processing and returns control to the OS.

```
conversion complete, 0 error(s) found.
```

o If any error other than those fatal is found, the structured assembler outputs the following message together with the number of errors found on the console on completion of its processing and returns control to the OS.

```
conversion complete, 5 error(s) found.
```

o If any fatal error which prohibits the structured assembler from continuing its processing is found, the structured assembler outputs the following message on the console, immediately terminates its processing, and returns control to the OS.

```
A>ST78Kn XXX.SRC
Structured assembler preprocessor for RA78Kn VX.X  [dd Mmm yy]
   Copyright (C) NEC Corporation 1988  US9876543210

A001 Missing input file

A>
```

7.4 Structured Assembler Options

7.4.1 Types of structured assembler options

The respective options of the structured assembler provide the structured assembler with detailed instructions or directions on its operation. These options are available in the following types:

Table 7-2. Structured Assembler Options

	Option name		Function
(1)	Processor type specification	C	Specifies the processor type of the target device subject to preprocessing. (See Note below.)
(2)	Word symbol character specification	SC	Specifies the last character of a word symbol.
(3)	Symbol definition	D	Specifies symbol(s) to be given to **#ifdef** directive.
(4)	No. of tab stops specification	WT	Specifies the output positions of converted instructions.
(5)	Include file specification	I	Specifies the drive and directory of an Include file.
(6)	Output file specification	O	Specifies the filename of a source file to be output.
(7)	Error list file specification	E	Specifies the filename of an error list to be output.
(8)	Parameter file specification	F	Specifies the filename of a parameter file to be input.

Note: This option is applicable only to the ST78K3.

7.4.2 How to specify options

(1) Option mark

Any option mark may be determined by setting a desired character in the environment variable "OPTMARK". The default character of this option mark is "-".

If two or more characters are set in the environment variable "OPTMARK", only the first character is accepted as a valid option mark.

(2) Option name

An option name may be described in uppercase or lowercase letters, because the structured assembler makes no distinction between the two. However, each option name must follow the option mark without space between the two.

(3) Option specification location in command line

Option(s) may be specified before or after the input file specification in the start-up command line of the structured assembler.

(4) Option parameter(s)

The parameter(s) to be given to each option must be described immediately after the option without spacing.

(5) Duplicate option specifications

If the same named options are specified two or more times, the last specified option becomes valid.

7.4.3 Description of each option

The description format, function, and application example(s) of each option are detailed below.

(1) **C** (Processor type specification)

[Description format]

```
-Cprocessor-type
```

[Function]

Specifies the processor type indicating the target device subject to preprocessing by the structured assembler. This option is applicable only to the ST78K3.

[Explanation]

With the ST78K3, one of the proccessor types indicating the respective target devices shown in Table 7-1 must be specified with the "C" option.

The structured assembler performs preprocessing on the target device specified by this option to generate source codes for the assembler.

If the **C** option is omitted, the structured assembler will output an error message.

Table 7-1. Target Devices

Processor type	Target device name
310	uPD78310
312	uPD78312
310A	uPD78310A
312A	uPD78312A
320	uPD78320
322	uPD78322
327	uPD78327
328	uPD78328
330	uPD78330
334	uPD78334

[Application example]

<How to specify C option in start-up command line>

```
A>ST78K3 TEST.S -C310
     ..................

conversion complete, 0 error(s) found

A>
```

(2) SC (Word symbol character specification)

[Description format]

```
-SCcharacter
```

[Function]

Specifies the last character of a symbol name representing a word symbol.

[Explanation]

The structured assembler generates different instructions depending on whether the data to be handled is a byte symbol or word symbol. In substitution, if the data is a byte symbol, the structured assembler generates a MOV instruction. If it is a word symbol reserved word (rp, sfrp, etc. of AX or other register pairs), the structured assembler generates a MOVW instruction. (See Section 2.3, "Reserved Words" in Chapter 2.)
If a symbol which is not the reserved word of the structured assembler is specified, the structured assembler generates instruction(s) by determining whether the symbol is a word symbol or byte symbol according to the last character of the symbol name.
If the SC option is omitted, the structured assembler assumes that the symbol ending with the character "P" or "p" is a word symbol.

[Application example]

<TEST.S>

```
A = #3
AX = #3
SYM = #3
SYM@ = #3
```

<How to specify SC option in start-up command line>

```
A>ST78K2 TEST.S -SC@
```

<TEST.ASM>

```
;A = #3
                MOV     A, #3
;AX = #3
                MOVW    AX, #3
;SYM = #3
                MOV     SYM, #3
;SYM@ = #3
                MOVW    SYM@, #3
```

(3) D (Symbol definition)

[Description format]

```
-Didentifier[=value]
```

[Function]

Defines an identifier.

[Explanation]

The D option defines an identifier. The value to be given to the identifier must be a binary, octal, decimal, or hexadecimal number. If the parameter "=value" is omitted, a value "1" is assumed to have been specified.

[Application example]

<TEST.S>

```
#ifdef TRUE
    text 1
#else
    text 2
#endif
```

<How to specify D option in start-up command line>

```
A>ST78K2 TEST.S -DTRUE=1
```

<TEST.ASM>

```
    text 1
```

(4) **WT** (No. of tab stops specification)

[Description format]

```
-WTvalue1, value2, value3
```

[Function]

Specifies the number of tab stops before the converted assembly language is to be output in each field.

[Explanation]

① Parameter "value1" specifies the number of tab stops before the output of the SYMBOL field. (Default value = 5)
Parameter "value2" specifies the number of tab stops before the output of the MNEMONIC field. (Default value = 6)
Parameter "value3" specifies the number of tab stops before the output of the OPERAND field. (Default value = 7)

② The above three parameters must be specified to satisfy the following relationship:

value1 < value2 <value3

[Application example]

<TEST.S>

```
A = #0
if( A == #1H )
    B = #0C5H
endif
```

<How to specify WT option in start-up command line>

```
A>ST78K2 TEST.S -WT3,4,5
```

<TEST.ASM>

```
;A = #0
                                MOV     A,#0
;if( A == #1H )
                                CMP     A,#1H
                                BNZ     $??1
;    B = #0C5H
                                MOV     B,#0C5H
;endif
                            ??1

*-------*-------*-------*-------*-------*
                    3       4       5   No. of tab stops
```

(5) I (Include file specification)

[Description format]

```
-I[drive-number:]directory
```

[Function]

Specifies the Include file which becomes an input to the structured assembler.

[Explanation]

① The I option specifies the drive number and directory name in which the Include file exists.

② If the I option is omitted, the structured assembler assumes that the Include file exists in the current directory on the current drive.

[Application example]

TEST.S Exists in the same directory as the structured assembler.

FILE Exists under the directory "INCLUDE" on drive B.

<TEST.S>

```
#include    "FILE"

        A=SIZE1
        B=SIZE2
```

<FILE>

```
#define    SIZE1 #08H
#define    SIZE2 #0AH
```

<How to specify I option in start-up command line>

```
A>ST78K2 TEST.S -IB:INCLUDE
```

<TEST.ASM>

```
;A=SIZE1
                    MOV     A,#08H
;B=SIZE2
                    MOV     B,#0AH
```

(6) O (Output file specification)

[Description format]

```
-O[[drive-number:]directory]filename
```

[Function]

Specifies the destination and filename of an output file.

[Explanation]

① The O option specifies the drive number, directory name, and filename of a secondary source file to be output after conversion of symbolic instructions to object codes.

② If the O option is omitted, the structured assembler generates an output file in the current directory by giving the input filename to the output file but with its file type (extension) changed to ".ASM".

③ "NUL" or "AUX" may be specified as "filename".

[Application example]

<TEST.S>

```
if( A == #1H )
    AX = #0C5H
endif
```

<How to specify O option in start-up command line>

```
A>ST78K2 TEST.S -OSAMPLE.ASM
```

<SAMPLE.ASM>

```
    ;if( A == #1H )
                                    CMP     A,#1H
                                    BNZ     $??1
    ;     AX = #0C5H
                                    MOV     AX,#0C5H
    ;endif
                              ??1:
```

(7) E (Error list file specification)

[Description format]

```
-E[[drive-number:]directory]filename
```

[Function]

Specifies the output destination and filename of an error list file.

[Explanation]

① The E option specifies the drive number, directory name, and filename of an error list file to be output.

② If the E option is omitted, the structured assembler generates an error list file in the current directory by giving the input filename to the error list file but with its file type (extension) changed to ".EST".

③ "NUL" or "AUX" may be specified as "filename".

[Application example]

<TEST.S>

```
if( A == #1H )
    AX = #0C5H
```

<How to specify E option in start-up command line>

```
A>ST78K2 TEST.S -ESAMPLE.EST

     ..................

missing endif

A>
```

<SAMPLE.ASM>

```
missing endif  TEST.S
conversion complete, 1 error(s) found
```

(8) F (Parameter file specification)

[Description format]

```
-F[[drive-number:]directory]filename
```

[Function]

Specifies the input drive number, directory, and filename of a parameter file.

[Explanation]

① The F option specifies the drive number, directory name, and filename of a parameter file to be input.

② With the F option, parameter "filename" cannot be omitted. If the file type (extension) is omitted from the filename specification, ".PST" is assumed as the file type.

[Application example]

<TEST.S>

```
if( A == #1H )
    AX = #0C5H
endif
```

<SAMPLE.PST>

```
TEST.S -ESAMPLE.EST
```

<How to specify F option in start-up command line>

```
A>ST78K2 -FSAMPLE.PST
     ..................

conversion complete, 0 error(s) found

A>
```

As a result of the above command input, TEST.S will be subjected to preprocessing by the structured assembler and a secondary source file and an error list file will be created in TEST.ASM and SAMPLE.EST, respectively.

<SAMPLE.EST>

```
conversion complete, 1 error(s) found
```

CHAPTER 8. INPUT/OUTPUT FILES

The structured assembler has the following three types of input files:

- o Input source module file
- o Include file
- o Parameter file (see Chapter 7)

The structured assembler also has the following two types of output files:

- o Secondary source module file
- o Error list file

8.1 Input Source Module File

The input/output files of the structured assembler are files which contain the assembly language described in the structured assembly language subject to preprocessing by the structured assembler.

Fig. 8-1 shows an example of an input source module file.

```
if( A == #0 )
    TMOD0 = BC
    BC = #0CH
else
    BC = #0AH
endif
TM0 = XA
CALL    !XXX
```

Fig. 8-1. Example of Input Source Module

8.2 Include File

8.2.1 What is an Include file?

The contents of a file separate from an input source module file can be expanded as is (without change) into the input source module file. This separate file is called an Include file.
By storing highly versatile symbol declarations in an Include file, the contents of the Include file can be read into two or more input source modules (programs).

8.2.2 Usage of Include file

The contents of an Include file will be expanded at the location of the **#include** directive described in the source module.
Fig. 8-2 shows an example of an input source module file containing a **#include** directive and an example of an Include file.

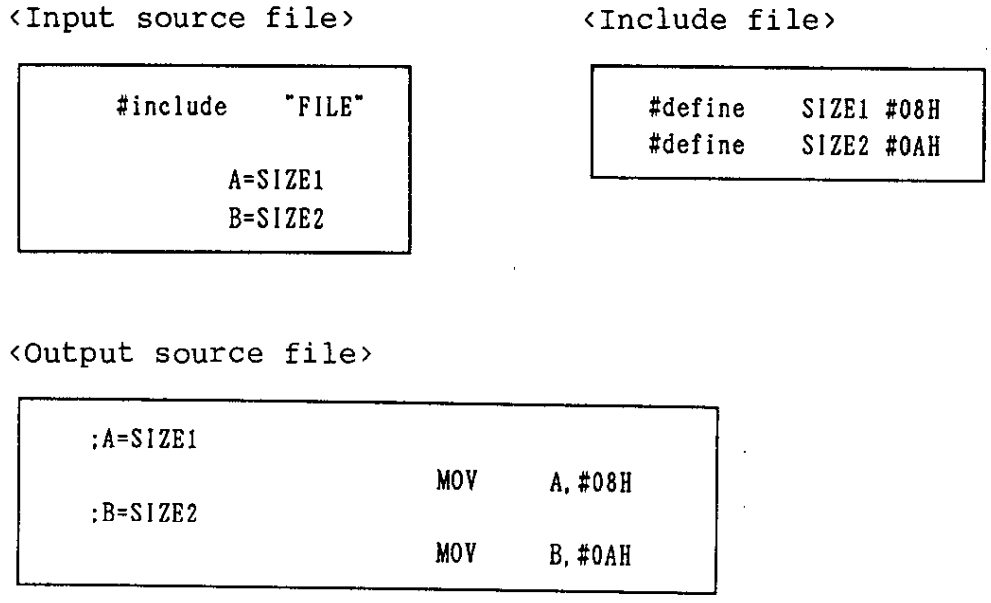


Fig. 8-2. Examples of Input Source Module File and Include File

8.3 Secondary Source Module File

A secondary source module file is a source module file which is output by the structured assembler and becomes a source module to be input to the assembler.

The secondary source module file will be expanded as shown below.

Input source module file	Secondary source module file
Control statements of structured assembler Expression statements of structured assembler	These statements will be output as comments.
Directives of structured assembler	These directives will not be output.
Comment statements	Comment statements will be output without change.
Other lines	Other lines will be output without change.

Fig. 8-3 shows an example of expanding a secondary source module file.

<Input source file>

```
#include    "FILE"
    if( A == #0 )
        TM0D0 = #0CH
    endif
    CALL      !XXX
    A=SIZE1
    B=SIZE2
```

<Include file>

```
#define    SIZE1 #08H
#define    SIZE2 #0AH
```

<Secondary source module file>

```
;if( A == #0 )
                          CMP     A,#0
                          BNZ     $??1
:    TM0D0 = #0CH
                          MOV     TM0D0,#0CH
;endif
                    ??1:
CALL      !XXX
;A=SIZE1
                          MOV     A,#08H
;B=SIZE2
                          MOV     B,#0AH
```

Fig. 8-3. Example of Secondary Source File Expansion

8.4 Error List File

An error list file is a file which stores error messages to be output when the structured assembler is started up.

Fig. 8-4 shows an example of an error list file.

<TEST.S>

```
if( A == #1H )
    AX = #0C5H
```

<Start-up of structured assembler>

```
A>ST78K2 TEST.S -ESAMPLE.EST

     ..................

missing endif

A>
```

<SAMPLE.EST>

```
missing endif  TEST.S
conversion complete, 1 error(s) found
```

Fig. 8-4. Example of Error List File

CHAPTER 9. ERROR MESSAGES AND TERMINATION INFORMATION

9.1 Error Messages

9.1.1 Fatal error messages

A fatal error message is output when program execution becomes impossible at the start-up of the structured assembler or when the structured assembler cannot continue its processing. On output of this error message, the structured assembler returns control to the OS.

Message	A001 Missing input file
Cause	No input file has been specified.
Program action	The structured assembler stops program execution.
User action	Specify the input file.

Message	A002 Too many input files
Cause	Two or more input filed have been specified.
Program action	The structured assembler stops program execution.
User action	Specify only one input file.

Message	A004 Illegal file name '<u>filename</u>'
Cause	The specified filename contains an illegal character or the number of characters used for the filename exceeded the limit value.
Program action	The structured assembler stops program execution.
User action	Specify the filename with correct characters or without exceeding the limit value for the number of characters.

Message	A005 Illegal file specification 'filename'
Cause	The specified file is illegal.
Program action	The structured assembler stops program execution.
User action	Specify the correct filename.

Message	A006 File not found 'filename'
Cause	The specified input file does not exist.
Program action	The structured assembler stops program execution.
User action	Specify an existing filename as the input file.

Message	A008 File specification conflicted 'filename'
Cause	The input and output filenames were specified in duplication.
Program action	The structured assembler stops program execution.
User action	Specify input and output filename different from each other.

Message	A009 Unable to make file 'filename'
Cause	The specified file has been write-protected.
Program action	The structured assembler stops program execution.
User action	Release the write-protect specification of the file.

Message	A010 Directory not found 'filename'
Cause	The specified output filename contains a drive number or directory name which does not exist.
Program action	The structured assembler stops program execution.
User action	Specify an existing drive number or directory name for the output file.

Message	A011 Illegal path 'option'
Cause	Other than a pathname has been specified for the displayed option which must specify a path as its parameter.
Program action	The structured assembler stops program execution.
User action	Specify the correct pathname for the option.

Message	A012 Missing parameter 'option'
Cause	The parameter required for the displayed option has not been specified.
Program action	The structured assembler stops program execution.
User action	Specify the required parameter for the option.

Message	A014 Out of range 'option'
Cause	The parameter value specified for the displayed option is outside the authorized value range.
Program action	The structured assembler stops program execution.
User action	Specify the correct value for the option.

Message	A015 Parameter is too long 'option'
Cause	The number of characters used to describe the parameter of the displayed option exceeded the limit value.
Program action	The structured assembler stops program execution.
User action	Specify the parameter without exceeding the limit value for the number of characters.

Message	A016 Illegal parameter 'option'
Cause	A syntax error exists in the parameter description of the displayed option.
Program action	The structured assembler stops program execution.
User action	Specify the correct parameter.

Message	A017 Too many parameters 'option'
Cause	The total number of parameters specified for the displayed option exceeded the limit value.
Program action	The structured assembler stops program execution.
User action	Keep the total number of parameters within the limit.

Message	A018 Option is not recognized 'option'
Cause	The specified option name is incorrect.
Program action	The structured assembler stops program execution.
User action	Specify the correct option name.

Message	A019 Parameter file nested
Cause	The F option has been specified in a parameter file.
Program action	The structured assembler stops program execution.
User action	Do not specify the F option in a parameter file.

Message	A020 Parameter file read error 'filename'
Cause	The displayed parameter file contains an illegal code.
Program action	The structured assembler stops program execution.
User action	Specify the correct parameter file.

Message	A021 Memory allocation failed
Cause	Memory space is insufficient.
Program action	The structured assembler stops program execution.
User action	Reserve the required memory space.

Message	A051 No processor specified
Cause	The device model number of the target device has not been specified.
Program action	The structured assembler stops program execution.
User action	Specify the device model number of the target device with the C option.

Message	A052 Illegal processor type specified
Cause	The specified device model number is incorrect.
Program action	The structured assembler stops program execution.
User action	Specify the correct device model number of the target device with the C option.

Message	cannot define the reserved symbol
Cause	A reserved word has been specified with the D option.
Program action	The structured assembler stops program execution.
User action	Do not specify any reserved word with the D option.

Message	cannot find 'filename'
Cause	An error exists in the file specification of the displayed Include file.
Program action	The structured assembler stops program execution.
User action	Specify the correct pathname, directory, and filename.

Message	illegal(-sc) character
Cause	A character which cannot be used as a symbol has been described in the SC option specification.
Program action	The structured assembler stops program execution.
User action	Describe the correct character in the SC option specification.

Message	read/write error on 'filename'
Cause	No free disk area exists.
Program action	The structured assembler stops program execution.
User action	Create a free disk area to read/write the file.

Message	symbol table overflow
Cause	The number of identifiers exceeded the limit value.
Program action	The structured assembler stops program execution.
User action	Reduce the number of identifiers.

Message	too many callt definition
Cause	The number of registered CALLT instructions exceeded the limit value.
Program action	The structured assembler stops program execution.
User action	Reduce the number of registered CALLT instructions.

9.1.2 Ordinary error messages

Ordinary error messages are output to a standard output device and an error list file if any syntax or other errors exist in the source program description. The structured assembler continues its processing after the output of any of these error message.

<Output format>

```
filename(nnnn)  message  source line display
```

nnnn: indicates the line number of the source file.

As "filename", a directory name is also output. In case of an Include file, the line number of the file is reset to "1" and then returns to the original number when the original file is restored.

Ordinary error messages are as listed below.

Message	duplicate definition
Cause	The same call instruction has been specified by **#defcallt** directive.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Correct the label registration of the **#defcallt** directive.

Message	illegal break
Cause	The **break** statement has been described in the incorrect position (in other than the **while, repeat, for,** or **switch** statement).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **break** statement in the correct position.

Message	illegal case/default/ends
Cause	The **case**, **default**, or **ends** statement has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **case**, **default**, or **ends** statement in the correct position.

Message	illegal continue
Cause	The **continue** statement has been described in the incorrect position (in other than the **while**, **repeat**, or **for** statement).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **continue** statement in the correct position.

Message	illegal elseif/else/endif
Cause	The **elseif**, **else**, or **endif** statement has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **elseif**, **else**, or **endif** statement in the correct position.

Message	illegal endw
Cause	The **endw** statement has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **endw** statement in the correct position.

Message	illegal next
Cause	The **next** statement has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **next** statement in the correct position.

Message	illegal register specified
Cause	In the Multiply & Substitute or Divide & Substitute expression statement, an incorrect register has been specified.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Confirm the register that can be specified.

Message	illegal until/until_bit
Cause	The **until** or **until_bit** statement has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **until** or **until_bit** statement in the correct position.

Message	illegal #else/#endif
Cause	The **#else** or **#endif** directive has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **#else** or **#endif** directive in the correct position.

Message	illegal #endcallt
Cause	The **#endcallt** directive has been described in the incorrect position.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **#endcallt** directive in the correct position.

Message	missing endif
Cause	The **endif** statement is missing (from the **if** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **endif** statement in the correct position.

Message	missing ends
Cause	The **ends** statement is missing (from the **switch** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **ends** statement in the correct position.

Message	missing endw
Cause	The **endw** statement is missing (from the **while** or **while_bit** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **endw** statement in the correct position.

Message	missing next
Cause	The **next** statement is missing (from the **for** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **next** statement in the correct position.

Message	missing until/until_bit
Cause	The **until** or **until_bit** statement is missing (from the **repeat** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **until** or **until_bit** statement in the correct position.

Message	missing #endcallt
Cause	The **#endcallt** directive is missing (from the **#defcallt** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **#endcallt** directive in the correct position.

Message	missing #endif
Cause	The **#endif** directive is missing (from the **#ifdef** block).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the **#endif** directive in the correct position.

Message	nest level error
Cause	A nesting error exists. (Nesting level exceeded the limit value).
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the correct control statement.

Message	syntax error
Cause	A syntax error exists in the described statement.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Describe the statement by following the syntax of the assembly language.

Message	too many callt instructions
Cause	Instructions to be defined in the **#defcallt-#endcallt** block are excessive.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Keep the number of instructions to be defined in the **#defcallt-#endcallt** block within the limit value.

Message	too many characters in a line
Cause	The length of one line exceeded the limit value.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Keep the length of one line within the limit value (120 characters max.).

Message	too many include files
Cause	**#include** directive exists in the Include file.
Program action	The structured assembler outputs the source line as is to the secondary source file.
User action	Do not write **#include** directive in an Include file.

9.1.3 Warning message

A warning message is not an error message but is output to alert the operator of that an undesirable process is being performed. Thus, the warning message is not counted as an error and the structured assembler accepts the process as valid and continues its processing.

<Output format>

```
filename(nnnn) warning:message source line display
```

Message	warning symbol redefinition
Cause	A symbol has been redefined by **#define** directive.
Program action	The structured assembler accepts the last defined symbol as valid.
User action	Correct the directive description if the first defined symbol should be accepted as valid.

9.2 Termination Information

The EXIT STATUS information to be returned to the OS when the structured assembler is terminated is shown below.

EXIT (0) ..... Normal termination
EXIT (1) ..... Abnormal termination (More than one error have been detected.)
EXIT (2) ..... Program has been aborted due to fatal error.

CHAPTER 10. LIST OF LIMIT VALUES

(1) Number of characters per line of source program:
120 characters (excluding LF or CR)

(2) Number of symbols (identifiers) that can be registered with #define directive:
512 symbols (excluding reserved words)

(3) Nesting level of control statements:
31 levels

(4) Nesting level of Conditional processing (#ifdef) directives:
8 levels

(5) Number of CALLT substituting (#defcallt) directives that can be described:
32 times per source program

(6) Nesting level of Include (#include) directives:
Nesting is not allowed.

LEGEND:

The symbols used in these appendixes and their meanings are as shown below.

o Types of Structured Assemblers

Symbol	Meaning
0	Applies to ST78K0.
I	Applies to ST78K1.
II	Applies to ST78K2.
III	Applies to ST78K3.
VI	Applies to ST78K6.

o Symbols in the Operand field

Symbol	Meaning
+	Autoincrement
#	Immediate data
!	Absolute address
$	Relative address
/	Bit inversion
[ ]	Indirect addressing
&	1-megabyte extended space

o Symbols with ST78K0/ST78K1

Representation format	Method of description
r	X(R0), A(R1), C(R2), B(R3), E(R4), D(R5,) L(R6), H(R7)
r1	A, B
r2	B, C
r3	D, E, E+
r4	D, E
rp	AX(RP0), BC(RP1), DE(RP2), HL(RP3)
sfr	Special function register symbol
sfrp	Special function register symbol (16-bit register)
saddr	FE20H to FF1FH; Immediate data or label
saddrp	FE20H to FF1FH: Immediate data (provided bit0=0) or label
addr16	0000H to FFFFH; Immediate data or label
addr13	0000H to 1FFFH; Immediate data or label
addr11	0800H to 0FFFH; Immediate data or label
addr5	0040H to 007EH; Immediate data or label
word	16-bit immediate data or label
byte	8-bit immediate data or label
bit	3-bit immediate data or label
n	3-bit immediate data

Note: With the uPD78112, saddr or saddrp becomes FE40H to FF1FH.

o Symbols with ST78K2

Representation format	Method of description
r	X(R0), A(R1), C(R2), B(R3), E(R4), D(R5,) L(R6), H(R7)
r1	B, C
rp	AX(RP0), BC(RP1), DE(RP2), HL(RP3)
sfr	Special function register symbol
sfrp	Special function register symbol (16-bit register)
mem	[DE], [HL],[DE+],[HL+],[DE-],[HL-], [DE+byte],[HL+byte],[SP+byte], word[A], word[B], word[DE], word[HL]
mem1	[DE],[HL]
saddr	FE20H to FF1FH; Immediate data or label
saddrp	FE20H to FF1EH: Immediate data (provided bit0=0) or label
addr16	0000H to FFFFH; Immediate data or label (with data access) 0000H to FE7FH; Immediate data or label (with BR instruction)
word	16-bit immediate data or label
byte	8-bit immediate data or label
bit	3-bit immediate data or label
n	3-bit immediate data

o Symbols with ST78K3

Representation format	Method of description
r	RSS=0: X(R0), A(R1), C(R2), B(R3), VP_L(R8), VP_H(R9), UP_L(R10), UP_H(R11), E(R12), D(R13), L(R14), H(R15) RSS=1: X(R4), A(R5), C(R6), B(R7), VP_L(R8), VP_H(R9), UP_L(R10), UP_H(R11), E(R12), D(R13), L(R14), H(R15)
r1	R0, R1, R2, R3, R4, R5, R6, R7
r2	R2, R3
rp	RSS=0: AX(RP0), BC(RP1), VP(RP4), UP(RP5), DE(RP6), HL(RP7) RSS=1: AX(RP2), BC(RP3), VP(RP4), UP(RP5), DE(RP6), HL(RP7)
rp1	RP0, RP1, RP2, RP3, RP4, RP5, RP6, RP7
rp2	DE, HL, VP, UP
sfr	Special function register symbol
sfrp	Special function register symbol (16-bit register)
mem	[DE], [HL],[DE+],[HL+],[DE-],[HL-], [VP], [UP], [DE+A], [HL+A], [DE+B], [HL+B], [VP+DE], [VP+HL], [DE+byte], [HL+byte], [VP+byte], [UP+byte], [SP+byte], word[A], word[B], word[DE], word[HL]
saddr	FE20H to FF1FH; Immediate data or label
saddrp	FE20H to FF1EH: Immediate data (provided bit0=0) or label
!addr16	0000H to FE7FH; Immediate data or label (provided addresses up to FFFF must be described with MOV instruction)
word	16-bit immediate data or label
byte	8-bit immediate data or label
bit	3-bit immediate data or label
n	3-bit immediate data (0 to 7)

o Symbols with ST78K6

Representation format	Method of descrition
br, br'	R0L, R0H, R1L, R1H, R2L, R2H, R3L, R3H R4L, R4H, R5L, R5H, R6L, R6H, R7L, R7H
br1	R0L, R0H, R1L, R1H, R2L, R2H, R3L, R3H
br2	R0L, R0H
wrL	R0L, R1L, R2L, R3L, R4L, R5L, R6L, R7L
wr, wr'	R0, R1, R2, R3, R4, R5, R6, R7
wr1	R0, R1, R2, R3
wr2	R4, R5, R6, R7
wr3	R0, R1
wr4	R6, R7
wrpL	R0, R2, R4, R6
wrp	RP0, RP1, RP2, RP3
bsfr	Special function register symbol
wsfr	Special function register symbol (16-bit register)
post	R0, R1, R2, R3, R4, R5, R6, R7 (Two or more registers may be described)
mem	[wr], [wr-], [wr+] : Register indirect addressing B:byte[wr], W:word[wr] : Based addressing [wr2](wr1) : Based indexed addressing with scaling B:byte[wr2](wr1), W:word[wr2](wr1): Based indexed addressing with scaling + displacement !addr16 : Direct addressing !addr16(wr) : Indexed addressing w/scaling
mem1	[wr2], [wr2-], [wr2+] : Register indirect addressing B:byte[wr2], W:word[wr2]: Based addressing !addr16 : Direct addressing
md	[R6+], [R6-]
ms	[R7+], [R7-]

Representation format	Method of description
bsaddr, bsaddr'	FC00H to FEFFH; Immediate data or label (8-bit manipulation)
wsaddr, wsaddr'	FC00H to FEFEH; Immediate data (provided bit0=0) or label (16-bit manipulation)
dsaddr	FC00H to FEFCH; Immediate data (provided bit0,1=0) or label (32-bit manipulation)
addr16	0000H to FFFFH; Immediate data or label
addr8	0000H to 00FFH: Immediate data or label
dword	32-bit immediate data or name
word	16-bit immediate data or name
byte	8-bit immediate data or name
bit4/bit3	4-/3-bit immediate data or name
n5/n4/n4	5-/4-/3-bit immediate data or name

APPENDIX A. LIST OF STATEMENT STRUCTURES

A-1. Control Statements

Control statement	Description format

if	**if** (condition expression 1) [register name]
	  if caluse
	elseif (condition expression 2) [register name]
	  elseif clause
	else
	  else clause
	endif

switch
 	
(see Note 1)
	switch (symbol)
	  case constant 1:
	  case 1 cause
	  case constant 2:
	  case 2 cause
	  :
	  case constant N:
	  case N cause
	  default:
	  default clause
	ends

for
 	
(see Note 2)
	for (expression 1;expression 2;expression 3)
	  [(register name)]
	  instruction group
	next

while	**while** (condition expression) [(register name)] instruction group **endw**
until	**repeat** instruction group **until** (condition expression) [(register name)]

Control statement	Description format
break	**break**
continue	**continue**
goto	**goto** label

if_bit	**if_bit** (bit condition expression 1)
	if clause
	elseif_bit (bit condition expression 2)
	elseif clause
	else
	else clause
	endif

while_bit	**while_bit** (bit condition expression) instruction group **endw**
until_bit	**repeat** instruction group **until_bit** (bit condition expression)

Note 1. The following symbols may be described in the **switch** statement.

Symbol	Operation	0	I	II	III	VI
r	if r=constant i, then goto clause i.	o	o	o	x	x
r1	if r1=constant i, then goto clause i.	x	x	x	o	x
br	if br=constant i, then goto clause i.	x	x	x	x	o
br1	if br1=constant i, then goto clause i.	x	x	x	x	o
[DE]	if (DE)=constant i, then goto clause i.	o	x	x	x	x
[HL]	if (HL)=constant i, then goto clause i.	o	o	x	x	x
[HL+byte]	if (HL+byte)=constant i, then goto clause i.	o	x	x	x	x
[HL+B]	if (HL+B)=constant i, then goto clause i.	o	x	x	x	x
[HL+C]	if (HL+C)=constant i, then goto clause i.	o	x	x	x	x
[r3]	if (FE00H+r3)=constant i, then goto clause i. (r3=00H to FFH)	x	o	x	x	x
word[r1]	if (word+r1)=constant i, then goto clause i.	x	o	x	x	x
saddr	if (saddr)=constant i, then goto clause i.	o	o	o	o	x
bsaddr	if (bsaddr)=constant i, then goto clause i.	x	x	x	x	o
sfr	if sfr=constant i, then goto clause i.	o	o	o	o	x
bsfr	if bsfr=constant i, then goto clause i.	x	x	x	x	o
mem	if (mem)=constant i, then goto clause i.	x	x	o	o	o
&mem	if (&mem)=constant i, then goto clause i.	x	x	o	x	x
[saddrp]	if (saddrp)=constant i, then goto clause i.	x	x	x	o	x
!addr16	if (!addr16)=constant i, then goto clause i.	o	x	o	o	x
&!addr16	if (&!addr16)=constant i, then goto clause i.	x	x	o	x	x
PSW	if PSW=constant i, then goto clause i.	o	x	o	x	x
PSWH	if PSWH=constant i, then goto clause i.	x	x	x	o	x
PSWL	if PSWL=constant i, then goto clause i.	x	x	x	o	x
AX	if AX=constant i, then goto clause i.	o	o	o	o	x
rp	if rp=constant i, then goto clause i.	o	x	x	x	x
saddrp	if (saddrp)=constant i, then goto clause i.	o	o	o	o	x
sfrp	if sfrp=constant i, then goto clause i.	o	o	o	o	x
mem1	if (mem1)=constant i, then goto clause i.	x	x	o	x	o
&mem1	if (&mem1)=constant i, then goto clause i.	x	x	o	x	x
wr	if wr=constant i, then goto clause i.	x	x	x	x	o
wsaddrp	if (wsaddrp)=constant i, then goto clause i.	x	x	x	x	o
wsfrp	if (wsfrp)=constant i, then goto clause i.	x	x	x	x	o
wrp	if wrp=constant i, then goto clause i.	x	x	x	x	o
dsaddrp	if (dsaddrp)=constant i, then goto clause i.	x	x	x	x	o

Note 2. Symbols that can be used in **for** statement

α (alpha), β (beta), γ (gamma), and δ (delta) that can be described in the **for** statement are as shown below (provided expression 1 must be a substitution expression, expression 2 be a relational operator expression, and expression 3 be a counter (incrmenet/decrement) statement).

<78K0/Ⅰ/Ⅱ/Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
–	r	#byte	A	○	○	○	×
–	r	#byte	r	×	○	○	×
–	rl	#byte	rl	×	×	×	○
–	A	#byte	#byte	○	○	○	○
–	A	#byte	r	○	○	○	×
–	A	#byte	rl	×	×	×	○
–	A	#byte	saddr	○	○	○	○
–	A	#byte	sfr	×	○	○	○
–	A	#byte	[r4]	×	○	×	×
–	A	#byte	[HL]	○	○	×	×
–	A	#byte	!addr16	○	×	×	×
–	A	#byte	[HL+byte]	○	×	×	×
–	A	#byte	[HL + B]	○	×	×	×
–	A	#byte	[HL + C]	○	×	×	×
–	A	#byte	mem	×	×	○	○
–	A	#byte	&mem	×	×	○	×
–	r	r	r	×	○	○	×
–	A	r	#byte	○	○	○	×
–	A	r	r	○	○	○	×
–	A	r	saddr	○	○	○	×
–	A	r	sfr	×	○	○	×
–	A	r	[r4]	×	○	×	×
–	A	r	[HL]	○	○	×	×
–	A	r	!addr16	○	×	×	×
–	A	r	[HL+byte]	○	×	×	×
–	A	r	[HL + B]	○	×	×	×
–	A	r	[HL + C]	○	×	×	×
–	A	r	mem	×	×	○	×
–	A	r	&mem	×	×	○	×
–	A	rl	#byte	×	×	×	○
–	A	rl	rl	×	×	×	○
–	A	rl	saddr	×	×	×	○
–	A	rl	sfr	×	×	×	○
–	A	rl	mem	×	×	×	○
–	A	saddr	#byte	○	○	○	○
–	A	saddr	r	○	○	○	×
–	A	saddr	rl	×	×	×	○
–	A	saddr	saddr	○	○	○	○
–	A	saddr	sfr	×	○	○	○
–	A	saddr	[r4]	×	○	×	×
–	A	saddr	[HL]	○	○	×	×
–	A	saddr	!addr16	○	×	×	×
–	A	saddr	[HL+byte]	○	×	×	×
–	A	saddr	[HL + B]	○	×	×	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0／Ⅰ／Ⅱ／Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
–	A	saddr	[HL + C]	○	×	×	×
–	A	saddr	mem	×	×	○	○
–	A	saddr	&mem	×	×	○	×
–	A	sfr	#byte	○	○	○	○
–	A	sfr	rl	×	×	×	○
–	A	sfr	r	○	○	○	×
–	A	sfr	saddr	○	○	○	○
–	A	sfr	sfr	×	○	○	○
–	A	sfr	[r4]	×	○	×	×
–	A	sfr	[HL]	○	○	×	×
–	A	sfr	!addr16	○	×	×	×
–	A	sfr	[HL+byte]	○	×	×	×
–	A	sfr	[HL + B]	○	×	×	×
–	A	sfr	[HL + C]	○	×	×	×
–	A	sfr	mem	×	×	○	○
–	A	sfr	&mem	×	×	○	×
–	A	[r3]	#byte	×	○	×	×
–	A	[r3]	r	×	○	×	×
–	A	[r3]	saddr	×	○	×	×
–	A	[r3]	sfr	×	○	×	×
–	A	[r3]	[r4]	×	○	×	×
–	A	[r3]	[HL]	×	○	×	×
–	A	[DE]	#byte	○	×	×	×
–	A	[DE]	r	○	×	×	×
–	A	[DE]	saddr	○	×	×	×
–	A	[DE]	[HL]	○	×	×	×
–	A	[DE]	!addr16	○	×	×	×
–	A	[DE]	[HL+byte]	○	×	×	×
–	A	[DE]	[HL + B]	○	×	×	×
–	A	[DE]	[HL + C]	○	×	×	×
–	A	[HL]	#byte	○	○	×	×
–	A	[HL]	r	○	○	×	×
–	A	[HL]	saddr	○	○	×	×
–	A	[HL]	sfr	×	○	×	×
–	A	[HL]	[r4]	×	○	×	×
–	A	[HL]	[HL]	○	○	×	×
–	A	[HL]	!addr16	○	×	×	×
–	A	[HL]	[HL+byte]	○	×	×	×
–	A	[HL]	[HL + B]	○	×	×	×
–	A	[HL]	[HL + C]	○	×	×	×
–	A	word[rl]	#byte	×	○	×	×
–	A	word[rl]	r	×	○	×	×
–	A	word[rl]	saddr	×	○	×	×
–	A	word[rl]	sfr	×	○	×	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0/ I /II /III>

α	β	γ	δ	0	I	II	III
–	A	word[r1]	[r4]	×	○	×	×
–	A	word[r1]	[HL]	×	○	×	×
–	A	mem	#byte	×	×	○	○
–	A	mem	r	×	×	○	×
–	A	mem	r1	×	×	×	○
–	A	mem	saddr	×	×	○	○
–	A	mem	sfr	×	×	○	○
–	A	mem	mem	×	×	○	○
–	A	mem	&mem	×	×	○	×
–	A	&mem	#byte	×	×	○	×
–	A	&mem	r	×	×	○	×
–	A	&mem	saddr	×	×	○	×
–	A	&mem	sfr	×	×	○	×
–	A	&mem	mem	×	×	○	×
–	A	&mem	&mem	×	×	○	×
–	A	[saddrp]	#byte	×	×	×	○
–	A	[saddrp]	r1	×	×	×	○
–	A	[saddrp]	saddr	×	×	×	○
–	A	[saddrp]	sfr	×	×	×	○
–	A	[saddrp]	mem	×	×	×	○
–	A	!addr16	#byte	○	×	○	○
–	A	!addr16	r	○	×	○	×
–	A	!addr16	r1	×	×	×	○
–	A	!addr16	saddr	○	×	○	○
–	A	!addr16	sfr	×	×	○	○
–	A	!addr16	[HL]	○	×	×	×
–	A	!addr16	!addr16	○	×	×	×
–	A	!addr16	[HL+byte]	○	×	×	×
–	A	!addr16	[HL + B]	○	×	×	×
–	A	!addr16	[HL + C]	○	×	×	×
–	A	!addr16	mem	×	×	○	○
–	A	!addr16	&mem	×	×	○	×
–	A	&!addr16	#byte	×	×	○	×
–	A	&!addr16	r	×	×	○	×
–	A	&!addr16	saddr	×	×	○	×
–	A	&!addr16	sfr	×	×	○	×
–	A	&!addr16	mem	×	×	○	×
–	A	&!addr16	&mem	×	×	○	×
–	A	PSW	#byte	○	○	○	×
–	A	PSW	r	○	○	○	×
–	A	PSW	saddr	○	○	○	×
–	A	PSW	sfr	×	○	○	×
–	A	PSW	[r4]	×	○	×	×
–	A	PSW	[HL]	○	○	×	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0／Ⅰ／Ⅱ／Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
–	A	PSW	!addr16	○	×	×	×
–	A	PSW	[HL+byte]	○	×	×	×
–	A	PSW	[HL + B]	○	×	×	×
–	A	PSW	[HL + C]	○	×	×	×
–	A	PSW	mem	×	×	○	×
–	A	PSW	&mem	×	×	○	×
–	A	PSWH	#byte	×	×	×	○
–	A	PSWH	r1	×	×	×	○
–	A	PSWH	saddr	×	×	×	○
–	A	PSWH	sfr	×	×	×	○
–	A	PSWH	mem	×	×	×	○
–	A	PSWL	#byte	×	×	×	○
–	A	PSWL	r1	×	×	×	○
–	A	PSWL	saddr	×	×	×	○
–	A	PSWL	sfr	×	×	×	○
–	A	PSWL	mem	×	×	×	○
–	A	[HL+byte]	#byte	○	×	×	×
–	A	[HL+byte]	r	○	×	×	×
–	A	[HL+byte]	saddr	○	×	×	×
–	A	[HL+byte]	[HL]	○	×	×	×
–	A	[HL+byte]	!addr16	○	×	×	×
–	A	[HL+byte]	[HL+byte]	○	×	×	×
–	A	[HL+byte]	[HL + B]	○	×	×	×
–	A	[HL+byte]	[HL + C]	○	×	×	×
–	A	[HL + B]	#byte	○	×	×	×
–	A	[HL + B]	r	○	×	×	×
–	A	[HL + B]	saddr	○	×	×	×
–	A	[HL + B]	[HL]	○	×	×	×
–	A	[HL + B]	!addr16	○	×	×	×
–	A	[HL + B]	[HL+byte]	○	×	×	×
–	A	[HL + B]	[HL + B]	○	×	×	×
–	A	[HL + B]	[HL + C]	○	×	×	×
–	A	[HL + C]	#byte	○	×	×	×
–	A	[HL + C]	r	○	×	×	×
–	A	[HL + C]	saddr	○	×	×	×
–	A	[HL + C]	[HL]	○	×	×	×
–	A	[HL + C]	!addr16	○	×	×	×
–	A	[HL + C]	[HL+byte]	○	×	×	×
–	A	[HL + C]	[HL + B]	○	×	×	×
–	A	[HL + C]	[HL + C]	○	×	×	×
–	saddr	#byte	#byte	○	○	○	○
–	saddr	#byte	saddr	×	×	○	○
–	saddr	A	#byte	○	○	○	○
–	saddr	A	saddr	×	×	○	○

Note 2. Symbols that can be used in **for** statement (contd)

<78K0/ I /II /III>

α	β	γ	δ	0	I	II	III
–	saddr	saddr	#byte	×	×	○	○
–	saddr	saddr	saddr	×	×	○	○
–	rp2	#word	rp1	×	×	×	○
–	AX	#word	#word	○	○	○	×
–	AX	#word	rp	×	○	○	×
–	AX	#word	saddrp	×	○	○	×
–	AX	#word	sfrp	×	○	○	×
–	AX	rp	#word	×	○	○	×
–	AX	rp	rp	×	○	○	×
–	AX	rp	saddrp	×	○	○	×
–	AX	rp	sfrp	×	○	○	×
–	AX	saddrp	#word	×	○	○	×
–	AX	saddrp	rp	×	○	○	×
–	AX	saddrp	saddrp	×	○	○	×
–	AX	saddrp	sfrp	×	○	○	×
–	AX	sfrp	#word	×	○	○	×
–	AX	sfrp	rp	×	○	○	×
–	AX	sfrp	saddrp	×	○	○	×
–	AX	sfrp	sfrp	×	○	○	×
–	AX	mem1	#word	×	×	○	×
–	AX	mem1	rp	×	×	○	×
–	AX	mem1	saddrp	×	×	○	×
–	AX	mem1	sfrp	×	×	○	×
–	AX	&mem1	#word	×	×	○	×
–	AX	&mem1	rp	×	×	○	×
–	AX	&mem1	saddrp	×	×	○	×
–	AX	&mem1	sfrp	×	×	○	×
–	saddrp	#word	#word	×	×	×	○
–	saddrp	#word	saddrp	×	×	×	○
–	saddrp	AX	#word	×	×	×	○
–	saddrp	AX	saddrp	×	×	×	○
–	saddrp	saddrp	#word	×	×	×	○
–	saddrp	saddrp	saddrp	×	×	×	○
A	r	r	#byte	○	×	×	×
A	r	r	saddr	○	×	×	×
A	r	r	[HL]	○	×	×	×
A	r	r	!addr16	○	×	×	×
A	r	r	[HL+byte]	○	×	×	×
A	r	r	[HL + B]	○	×	×	×
A	r	r	[HL + C]	○	×	×	×
A	r1	#byte	#byte	×	×	×	○
A	r1	#byte	saddr	×	×	×	○
A	r1	#byte	sfr	×	×	×	○
A	r1	#byte	mem	×	×	×	○

Note 2. Symbols that can be used in **for** statement (contd)

<78K0/Ⅰ/Ⅱ/Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
A	r	saddr	#byte	○	○	○	×
A	r	saddr	saddr	○	○	○	×
A	r	saddr	sfr	×	○	○	×
A	r	saddr	[r4]	×	○	×	×
A	r	saddr	[HL]	○	○	×	×
A	r	saddr	!addr16	○	×	×	×
A	r	saddr	[HL+byte]	○	×	×	×
A	r	saddr	[HL + B]	○	×	×	×
A	r	saddr	[HL + C]	○	×	×	×
A	r	saddr	mem	×	×	○	×
A	r	saddr	&mem	×	×	○	×
A	r1	saddr	#byte	×	×	×	○
A	r1	saddr	saddr	×	×	×	○
A	r1	saddr	sfr	×	×	×	○
A	r1	saddr	mem	×	×	×	○
A	r	sfr	#byte	○	○	○	×
A	r	sfr	saddr	○	○	○	×
A	r	sfr	sfr	×	○	○	×
A	r	sfr	[r4]	×	○	×	×
A	r	sfr	[HL]	○	○	×	×
A	r	sfr	!addr16	○	×	×	×
A	r	sfr	[HL+byte]	○	×	×	×
A	r	sfr	[HL + B]	○	×	×	×
A	r	sfr	[HL + C]	○	×	×	×
A	r	sfr	mem	×	×	○	×
A	r	sfr	&mem	×	×	○	×
A	r1	sfr	#byte	×	×	×	○
A	r1	sfr	saddr	×	×	×	○
A	r1	sfr	sfr	×	×	×	○
A	r1	sfr	mem	×	×	×	○
A	r	[r3]	#byte	×	○	×	×
A	r	[r3]	saddr	×	○	×	×
A	r	[r3]	sfr	×	○	×	×
A	r	[r3]	[r4]	×	○	×	×
A	r	[r3]	[HL]	×	○	×	×
A	r	[DE]	#byte	○	×	×	×
A	r	[DE]	saddr	○	×	×	×
A	r	[DE]	[HL]	○	×	×	×
A	r	[DE]	!addr16	○	×	×	×
A	r	[DE]	[HL+byte]	○	×	×	×
A	r	[DE]	[HL + B]	○	×	×	×
A	r	[DE]	[HL + C]	○	×	×	×
A	r	[HL]	#byte	○	○	×	×
A	r	[HL]	saddr	○	○	×	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0／Ⅰ／Ⅱ／Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
A	r	[HL]	sfr	×	○	×	×
A	r	[HL]	[r4]	×	○	×	×
A	r	[HL]	[HL]	○	○	×	×
A	r	[HL]	!addr16	○	×	×	×
A	r	[HL]	[HL+byte]	○	×	×	×
A	r	[HL]	[HL + B]	○	×	×	×
A	r	[HL]	[HL + C]	○	×	×	×
A	r	word[rl]	#byte	×	○	×	×
A	r	word[rl]	saddr	×	○	×	×
A	r	word[rl]	sfr	×	○	×	×
A	r	word[rl]	[r4]	×	○	×	×
A	r	word[rl]	[HL]	×	○	×	×
A	r	mem	#byte	×	×	○	×
A	r	mem	saddr	×	×	○	×
A	r	mem	sfr	×	×	○	×
A	r	mem	mem	×	×	○	×
A	r	mem	&mem	×	×	○	×
A	rl	mem	#byte	×	×	×	○
A	rl	mem	saddr	×	×	×	○
A	rl	mem	sfr	×	×	×	○
A	rl	mem	mem	×	×	×	○
A	rl	[saddrp]	#byte	×	×	×	○
A	rl	[saddrp]	saddr	×	×	×	○
A	rl	[saddrp]	sfr	×	×	×	○
A	rl	[saddrp]	mem	×	×	×	○
A	r	&mem	#byte	×	×	○	×
A	r	&mem	saddr	×	×	○	×
A	r	&mem	sfr	×	×	○	×
A	r	&mem	mem	×	×	○	×
A	r	&mem	&mem	×	×	○	×
A	r	!addr16	#byte	○	×	○	×
A	r	!addr16	saddr	○	×	○	×
A	r	!addr16	sfr	×	×	○	×
A	r	!addr16	[HL]	○	×	×	×
A	r	!addr16	!addr16	○	×	×	×
A	r	!addr16	[HL+byte]	○	×	×	×
A	r	!addr16	[HL + B]	○	×	×	×
A	r	!addr16	[HL + C]	○	×	×	×
A	r	!addr16	mem	×	×	○	×
A	r	!addr16	&mem	×	×	○	×
A	rl	!addr16	#byte	×	×	×	○
A	rl	!addr16	saddr	×	×	×	○
A	rl	!addr16	sfr	×	×	×	○
A	rl	!addr16	mem	×	×	×	○

Note 2. Symbols that can be used in **for** statement (contd)

<78K0／Ⅰ／Ⅱ／Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
A	r	&!addr16	#byte	×	×	○	×
A	r	&!addr16	saddr	×	×	○	×
A	r	&!addr16	sfr	×	×	○	×
A	r	&!addr16	mem	×	×	○	×
A	r	&!addr16	&mem	×	×	○	×
A	r	[HL+byte]	#byte	○	×	×	×
A	r	[HL+byte]	saddr	○	×	×	×
A	r	[HL+byte]	[HL]	○	×	×	×
A	r	[HL+byte]	!addr16	○	×	×	×
A	r	[HL+byte]	[HL+byte]	○	×	×	×
A	r	[HL+byte]	[HL + B]	○	×	×	×
A	r	[HL+byte]	[HL + C]	○	×	×	×
A	r	[HL + B]	#byte	○	×	×	×
A	r	[HL + B]	saddr	○	×	×	×
A	r	[HL + B]	[HL]	○	×	×	×
A	r	[HL + B]	!addr16	○	×	×	×
A	r	[HL + B]	[HL+byte]	○	×	×	×
A	r	[HL + B]	[HL + B]	○	×	×	×
A	r	[HL + B]	[HL + C]	○	×	×	×
A	r	[HL + C]	#byte	○	×	×	×
A	r	[HL + C]	saddr	○	×	×	×
A	r	[HL + C]	[HL]	○	×	×	×
A	r	[HL + C]	!addr16	○	×	×	×
A	r	[HL + C]	[HL+byte]	○	×	×	×
A	r	[HL + C]	[HL + B]	○	×	×	×
A	r	[HL + C]	[HL + C]	○	×	×	×
A	r	PSW	#byte	○	○	○	×
A	r	PSW	saddr	○	○	○	×
A	r	PSW	sfr	×	○	○	×
A	r	PSW	[r4]	×	○	×	×
A	r	PSW	[HL]	○	○	×	×
A	r	PSW	!addr16	○	×	×	×
A	r	PSW	[HL+byte]	○	×	×	×
A	r	PSW	[HL + B]	○	×	×	×
A	r	PSW	[HL + C]	○	×	×	×
A	r	PSW	mem	×	×	○	×
A	r	PSW	&mem	×	×	○	×
A	rl	PSWH	#byte	×	×	×	○
A	rl	PSWH	saddr	×	×	×	○
A	rl	PSWH	sfr	×	×	×	○
A	rl	PSWH	mem	×	×	×	○
A	rl	PSWL	#byte	×	×	×	○
A	rl	PSWL	saddr	×	×	×	○
A	rl	PSWL	sfr	×	×	×	○

Note 2. Symbols that can be used in **for** statement (contd)

<78K0/Ⅰ/Ⅱ/Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
A	rl	PSWL	mem	×	×	×	○
A	saddr	r	r	×	○	○	×
A	saddr	r	saddr	○	○	×	×
A	saddr	r	sfr	×	○	○	×
A	saddr	r	[r4]	×	○	×	×
A	saddr	r	[HL]	○	○	×	×
A	saddr	r	!addr16	○	×	×	×
A	saddr	r	[HL+byte]	○	×	×	×
A	saddr	r	[HL + B]	○	×	×	×
A	saddr	r	[HL + C]	○	×	×	×
A	saddr	r	mem	×	×	○	×
A	saddr	r	&mem	×	×	○	×
A	saddr	rl	rl	×	×	×	○
A	saddr	rl	sfr	×	×	×	○
A	saddr	rl	mem	×	×	×	○
A	saddr	saddr	r	×	○	×	×
A	saddr	saddr	saddr	○	○	×	×
A	saddr	saddr	sfr	×	○	×	×
A	saddr	saddr	[r4]	×	○	×	×
A	saddr	saddr	[HL]	○	○	×	×
A	saddr	saddr	!addr16	○	×	×	×
A	saddr	saddr	[HL+byte]	○	×	×	×
A	saddr	saddr	[HL + B]	○	×	×	×
A	saddr	saddr	[HL + C]	○	×	×	×
A	saddr	sfr	r	×	○	○	×
A	saddr	sfr	rl	×	×	×	○
A	saddr	sfr	saddr	○	○	×	×
A	saddr	sfr	sfr	×	○	○	○
A	saddr	sfr	[r4]	×	○	×	×
A	saddr	sfr	[HL]	○	○	×	×
A	saddr	sfr	!addr16	○	×	×	×
A	saddr	sfr	[HL+byte]	○	×	×	×
A	saddr	sfr	[HL + B]	○	×	×	×
A	saddr	sfr	[HL + C]	○	×	×	×
A	saddr	sfr	mem	×	×	○	○
A	saddr	sfr	&mem	×	×	○	×
A	saddr	[r3]	r	×	○	×	×
A	saddr	[r3]	saddr	×	○	×	×
A	saddr	[r3]	sfr	×	○	×	×
A	saddr	[r3]	[r4]	×	○	×	×
A	saddr	[r3]	[HL]	×	○	×	×
A	saddr	[DE]	saddr	○	×	×	×
A	saddr	[DE]	[HL]	○	×	×	×
A	saddr	[DE]	!addr16	○	×	×	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0/Ⅰ/Ⅱ/Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
A	r1	[DE]	[HL+byte]	○	×	×	×
A	saddr	[DE]	[HL + B]	○	×	×	×
A	saddr	[DE]	[HL + C]	○	×	×	×
A	saddr	[HL]	r	×	○	×	×
A	saddr	[HL]	saddr	○	○	×	×
A	saddr	[HL]	sfr	×	○	×	×
A	saddr	[HL]	[r4]	×	○	×	×
A	saddr	[HL]	[HL]	○	○	×	×
A	saddr	[HL]	!addr16	○	×	×	×
A	saddr	[HL]	[HL+byte]	○	×	×	×
A	saddr	[HL]	[HL + B]	○	×	×	×
A	saddr	[HL]	[HL + C]	○	×	×	×
A	saddr	word[r1]	r	×	○	×	×
A	saddr	word[r1]	saddr	×	○	×	×
A	saddr	word[r1]	sfr	×	○	×	×
A	saddr	word[r1]	[r4]	×	○	×	×
A	saddr	word[r1]	[HL]	×	○	×	×
A	saddr	mem	r	×	×	○	×
A	saddr	mem	r1	×	×	×	○
A	saddr	mem	sfr	×	×	○	○
A	saddr	mem	mem	×	×	○	○
A	saddr	mem	&mem	×	×	○	×
A	saddr	[saddrp]	r1	×	×	×	○
A	saddr	[saddrp]	sfr	×	×	×	○
A	saddr	[saddrp]	mem	×	×	×	○
A	saddr	&mem	r	×	×	○	×
A	saddr	&mem	sfr	×	×	○	×
A	saddr	&mem	mem	×	×	○	×
A	saddr	&mem	&mem	×	×	○	×
A	saddr	!addr16	r	×	×	○	×
A	saddr	!addr16	r1	×	×	×	○
A	saddr	!addr16	saddr	○	×	×	×
A	saddr	!addr16	sfr	×	×	○	○
A	saddr	!addr16	[HL]	○	×	×	×
A	saddr	!addr16	!addr16	○	×	×	×
A	saddr	!addr16	[HL+byte]	○	×	×	×
A	saddr	!addr16	[HL + B]	○	×	×	×
A	saddr	!addr16	[HL + C]	○	×	×	×
A	saddr	!addr16	mem	×	×	○	○
A	saddr	!addr16	&mem	×	×	○	×
A	saddr	&!addr16	r	×	×	○	×
A	saddr	&!addr16	sfr	×	×	○	×
A	saddr	&!addr16	mem	×	×	○	×
A	saddr	&!addr16	&mem	×	×	○	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0/Ⅰ/Ⅱ/Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
A	r1	[HL+byte]	saddr	○	×	×	×
A	saddr	[HL+byte]	[HL]	○	×	×	×
A	saddr	[HL+byte]	!addr16	○	×	×	×
A	saddr	[HL+byte]	[HL+byte]	○	×	×	×
A	saddr	[HL+byte]	[HL + B]	○	×	×	×
A	saddr	[HL+byte]	[HL + C]	○	×	×	×
A	saddr	[HL + B]	saddr	○	×	×	×
A	saddr	[HL + B]	[HL]	○	×	×	×
A	saddr	[HL + B]	!addr16	○	×	×	×
A	saddr	[HL + B]	[HL+byte]	○	×	×	×
A	saddr	[HL + B]	[HL + B]	○	×	×	×
A	saddr	[HL + B]	[HL + C]	○	×	×	×
A	saddr	[HL + C]	saddr	○	×	×	×
A	saddr	[HL + C]	[HL]	○	×	×	×
A	saddr	[HL + C]	!addr16	○	×	×	×
A	saddr	[HL + C]	[HL+byte]	○	×	×	×
A	saddr	[HL + C]	[HL + B]	○	×	×	×
A	saddr	[HL + C]	[HL + C]	○	×	×	×
A	saddr	PSW	r	×	○	○	×
A	saddr	PSW	saddr	○	○	×	×
A	saddr	PSW	sfr	×	○	○	×
A	saddr	PSW	[r4]	×	○	×	×
A	saddr	PSW	[HL]	○	○	×	×
A	saddr	PSW	!addr16	○	×	×	×
A	saddr	PSW	[HL+byte]	○	×	×	×
A	saddr	PSW	[HL + B]	○	×	×	×
A	saddr	PSW	[HL + C]	○	×	×	×
A	saddr	PSW	mem	×	×	○	×
A	saddr	PSW	&mem	×	×	○	×
A	saddr	PSWH	r1	×	×	×	○
A	saddr	PSWH	sfr	×	×	×	○
A	saddr	PSWH	mem	×	×	×	○
A	saddr	PSWL	r1	×	×	×	○
A	saddr	PSWL	sfr	×	×	×	○
A	saddr	PSWL	mem	×	×	×	○
AX	rp	saddrp	#word	○	○	○	×
AX	rp	saddrp	rp	×	○	○	×
AX	rp	saddrp	saddrp	×	○	○	×
AX	rp	saddrp	sfrp	×	○	○	×
AX	rp2	saddrp	saddrp	×	×	×	○
AX	rp2	saddrp	sfrp	×	×	×	○
AX	rp	sfrp	#word	○	○	○	×
AX	rp	sfrp	rp	×	○	○	×
AX	rp	sfrp	saddrp	×	○	○	×

Note 2. Symbols that can be used in **for** statement (contd)

<78K0／Ⅰ／Ⅱ／Ⅲ>

α	β	γ	δ	0	Ⅰ	Ⅱ	Ⅲ
AX	rp	sfrp	sfrp	×	○	○	○
AX	rp2	sfrp	saddrp	×	×	×	○
AX	rp2	sfrp	sfrp	×	×	×	○
AX	rp	mem1	#word	×	×	○	×
AX	rp	mem1	rp	×	×	○	×
AX	rp	mem1	saddrp	×	×	○	×
AX	rp	mem1	sfrp	×	×	○	×
AX	rp	&mem1	#word	×	×	○	×
AX	rp	&mem1	rp	×	×	○	×
AX	rp	&mem1	saddrp	×	×	○	×
AX	rp	&mem1	sfrp	×	×	○	×
AX	rp	rp	#word	○	×	×	×
AX	rp	!addr16	#word	○	×	×	×
AX	saddrp	rp1	rp1	×	×	×	○
AX	saddrp	rp1	sfrp	×	×	×	○
AX	saddrp	sfrp	rp1	×	×	×	○
AX	saddrp	sfrp	sfrp	×	×	×	○

Note 2. Symbols that can be used in **for** statement (contd)

<78K/VI>

α	β	γ	δ	α	β	γ	δ
–	br	#byte	#byte	–	meml	#byte	brl
–	br	#byte	br'	–	meml	#byte	#byte
–	br	#byte	mem	–	wr	#word	#word
–	brl	#byte	bsaddr	–	wr	#word	wr'
–	brl	#byte	bsfr	–	wr	#word	wsaddr
–	brl	#byte	meml	–	wr	#word	wsfr
–	br	br'	#byte	–	wr	#word	mem
–	br	br'	br'	–	wr	#word	meml
–	br	br'	mem	–	wr	wr'	#word
–	brl	br'	bsaddr	–	wr	wr'	wr'
–	brl	br'	bsfr	–	wr	wr'	wsaddr
–	brl	br'	meml	–	wr	wr'	wsfr
–	brl	bsaddr	bsaddr	–	wr	wr'	mem
–	brl	bsaddr	bsfr	–	wr	wr'	meml
–	brl	bsaddr	meml	–	wr	wsaddr	#word
–	brl	[wsaddr]	bsaddr	–	wr	wsaddr	wr'
–	brl	[wsaddr]	bsfr	–	wr	wsaddr	wsaddr
–	brl	[wsaddr]	meml	–	wr	wsaddr	wsfr
–	brl	bsfr	bsaddr	–	wr	wsaddr	mem
–	brl	bsfr	bsfr	–	wr	wsaddr	meml
–	brl	bsfr	meml	–	wr	[wsaddr]	#word
–	br	mem	#byte	–	wr	[wsaddr]	wr'
–	br	mem	br'	–	wr	[wsaddr]	wsaddr
–	br	mem	mem	–	wr	[wsaddr]	wsfr
–	brl	mem	bsaddr	–	wr	[wsaddr]	mem
–	brl	mem	bsfr	–	wr	[wsaddr]	meml
–	brl	mem	meml	–	wr	wsfr	#word
–	brl	meml	bsaddr	–	wr	wsfr	wr'
–	brl	meml	bsfr	–	wr	wsfr	wsaddr
–	brl	meml	meml	–	wr	wsfr	wsfr
–	bsaddr	#byte	#byte	–	wr	wsfr	mem
–	bsaddr	#byte	bsaddr'	–	wr	wsfr	meml
–	bsaddr	#byte	brl	–	wr	mem	#word
–	bsaddr	bsaddr'	#byte	–	wr	mem	wr'
–	bsaddr	bsaddr'	bsaddr'	–	wr	mem	wsaddr
–	bsaddr	bsaddr'	brl	–	wr	mem	wsfr
–	bsaddr	brl	#byte	–	wr	mem	mem
–	bsaddr	brl	bsaddr'	–	wr	mem	meml
–	bsaddr	brl	brl	–	wr	meml	#word
–	mem	br	br	–	wr	meml	wr'
–	mem	br	#byte	–	wr	meml	wsaddr
–	mem	#byte	br	–	wr	meml	wsfr
–	mem	#byte	#byte	–	wr	meml	mem
–	meml	brl	brl	–	wr	meml	meml
–	meml	brl	#byte	–	wr	PSW	#word

Note 2. Symbols that can be used in **for** statement (contd)

<78K/VI>

α	β	γ	δ	α	β	γ	δ
–	wr	PSW	wr'	brl	bsaddr	bsfr	bsfr
–	wr	PSW	wsaddr	brl	bsaddr	bsfr	mem
–	wr	PSW	wsfr	brl	bsaddr	bsfr	meml
–	wr	PSW	mem	brl	bsaddr	mem	br'
–	wr	PSW	meml	brl	bsaddr	mem	bsaddr
–	wr	SP	#word	brl	bsaddr	mem	bsfr
–	wr	SP	wr'	brl	bsaddr	mem	mem
–	wr	SP	wsaddr	brl	bsaddr	mem	meml
–	wr	SP	wsfr	brl	bsaddr	meml	br'
–	wr	SP	mem	brl	bsaddr	meml	bsaddr
–	wr	SP	meml	brl	bsaddr	meml	bsfr
–	wsaddr	#word	#word	brl	bsaddr	meml	mem
–	wsaddr	#word	wsaddr'	brl	bsaddr	meml	meml
–	wsaddr	#word	wr	br	mem	br'	br'
–	wsaddr	wsaddr'	#word	br	mem	br'	mem
–	wsaddr	wsaddr'	wsaddr'	br	mem	mem	br'
–	wsaddr	wsaddr'	wr	br	mem	mem	mem
–	wsaddr	wr	#word	brl	meml	br'	br'
–	wsaddr	wr	wsaddr'	brl	meml	br'	bsaddr
–	wsaddr	wr	wr	brl	meml	br'	bsfr
–	mem	wr	wr	brl	meml	br'	mem
–	mem	wr	#word	brl	meml	br'	meml
–	mem	#word	wr	brl	meml	bsaddr	br'
–	mem	#word	#word	brl	meml	bsaddr	bsaddr
–	meml	wr	wr	brl	meml	bsaddr	bsfr
–	meml	wr	#word	brl	meml	bsaddr	mem
–	meml	#word	wr	brl	meml	bsaddr	meml
–	meml	#word	#word	brl	meml	[wsaddr]	br'
brl	bsaddr	br'	br'	brl	meml	[wsaddr]	bsaddr
brl	bsaddr	br'	bsaddr	brl	meml	[wsaddr]	bsfr
brl	bsaddr	br'	bsfr	brl	meml	[wsaddr]	mem
brl	bsaddr	br'	mem	brl	meml	[wsaddr]	meml
brl	bsaddr	br'	meml	brl	meml	bsfr	br'
brl	bsaddr	bsaddr	br'	brl	meml	bsfr	bsaddr
brl	bsaddr	bsaddr	bsaddr	brl	meml	bsfr	bsfr
brl	bsaddr	bsaddr	bsfr	brl	meml	bsfr	mem
brl	bsaddr	bsaddr	mem	brl	meml	bsfr	meml
brl	bsaddr	bsaddr	meml	brl	meml	mem	br'
brl	bsaddr	[wsaddr]	br'	brl	meml	mem	bsaddr
brl	bsaddr	[wsaddr]	bsaddr	brl	meml	mem	bsfr
brl	bsaddr	[wsaddr]	bsfr	brl	meml	mem	mem
brl	bsaddr	[wsaddr]	mem	brl	meml	mem	meml
brl	bsaddr	[wsaddr]	meml	brl	meml	meml	br'
brl	bsaddr	bsfr	br'	brl	meml	meml	bsaddr
brl	bsaddr	bsfr	bsaddr	brl	meml	meml	bsfr

Note 2. Symbols that can be used in **for** statement (contd)

<78K/VI>

α	β	γ	δ	α	β	γ	δ
brl	meml	meml	mem	wr	SP	wr'	wsfr
brl	meml	meml	meml	wr	SP	wr'	mem
wr	wsaddr	wr'	wr'	wr	SP	wr'	meml
wr	wsaddr	wr'	wsaddr	wr	SP	wsaddr	#word
wr	wsaddr	wr'	wsfr	wr	SP	wsaddr	wr'
wr	wsaddr	wr'	mem	wr	SP	wsaddr	wsaddr
wr	wsaddr	wr'	meml	wr	SP	wsaddr	wsfr
wr	wsaddr	wsaddr	wr'	wr	SP	wsaddr	mem
wr	wsaddr	wsaddr	wsaddr	wr	SP	wsaddr	meml
wr	wsaddr	wsaddr	wsfr	wr	SP	[wsaddr]	#word
wr	wsaddr	wsaddr	mem	wr	SP	[wsaddr]	wr'
wr	wsaddr	wsaddr	meml	wr	SP	[wsaddr]	wsaddr
wr	wsaddr	[wsaddr]	wr'	wr	SP	[wsaddr]	wsfr
wr	wsaddr	[wsaddr]	wsaddr	wr	SP	[wsaddr]	mem
wr	wsaddr	[wsaddr]	wsfr	wr	SP	[wsaddr]	meml
wr	wsaddr	[wsaddr]	mem	wr	SP	wsfr	#word
wr	wsaddr	[wsaddr]	meml	wr	SP	wsfr	wr'
wr	wsaddr	wsfr	wr'	wr	SP	wsfr	wsaddr
wr	wsaddr	wsfr	wsaddr	wr	SP	wsfr	wsfr
wr	wsaddr	wsfr	wsfr	wr	SP	wsfr	mem
wr	wsaddr	wsfr	mem	wr	SP	wsfr	meml
wr	wsaddr	wsfr	meml	wr	SP	mem	#word
wr	wsaddr	mem	wr'	wr	SP	mem	wr'
wr	wsaddr	mem	wsaddr	wr	SP	mem	wsaddr
wr	wsaddr	mem	wsfr	wr	SP	mem	wsfr
wr	wsaddr	mem	mem	wr	SP	mem	mem
wr	wsaddr	mem	meml	wr	SP	mem	meml
wr	wsaddr	meml	wr'	wr	SP	meml	#word
wr	wsaddr	meml	wsaddr	wr	SP	meml	wr'
wr	wsaddr	meml	wsfr	wr	SP	meml	wsaddr
wr	wsaddr	meml	mem	wr	SP	meml	wsfr
wr	wsaddr	meml	meml	wr	SP	meml	mem
wr	wsaddr	PSW	wr'	wr	SP	meml	meml
wr	wsaddr	PSW	wsaddr	wr	SP	PSW	#word
wr	wsaddr	PSW	wsfr	wr	SP	PSW	wr'
wr	wsaddr	PSW	mem	wr	SP	PSW	wsaddr
wr	wsaddr	PSW	meml	wr	SP	PSW	wsfr
wr	wsaddr	SP	wr'	wr	SP	PSW	mem
wr	wsaddr	SP	wsaddr	wr	SP	PSW	meml
wr	wsaddr	SP	wsfr	wr	SP	SP	#word
wr	wsaddr	SP	mem	wr	SP	SP	wr'
wr	wsaddr	SP	meml	wr	SP	SP	wsaddr
wr	SP	wr'	#word	wr	SP	SP	wsfr
wr	SP	wr'	wr'	wr	SP	SP	mem
wr	SP	wr'	wsaddr	wr	SP	SP	meml

Note 2. Symbols that can be used in **for** statement (contd)

<78K/VI>

α	β	γ	δ	α	β	γ	δ
wr	mem	wr'	wr'	wr	meml	wsaddr	wr'
wr	mem	wr'	wsaddr	wr	meml	wsaddr	wsaddr
wr	mem	wr'	wsfr	wr	meml	wsaddr	wsfr
wr	mem	wr'	mem	wr	meml	wsaddr	mem
wr	mem	wr'	meml	wr	meml	wsaddr	meml
wr	mem	wsaddr	wr'	wr	meml	[wsaddr]	wr'
wr	mem	wsaddr	wsaddr	wr	meml	[wsaddr]	wsaddr
wr	mem	wsaddr	wsfr	wr	meml	[wsaddr]	wsfr
wr	mem	wsaddr	mem	wr	meml	[wsaddr]	mem
wr	mem	wsaddr	meml	wr	meml	[wsaddr]	meml
wr	mem	[wsaddr]	wr'	wr	meml	wsfr	wr'
wr	mem	[wsaddr]	wsaddr	wr	meml	wsfr	wsaddr
wr	mem	[wsaddr]	wsfr	wr	meml	wsfr	wsfr
wr	mem	[wsaddr]	mem	wr	meml	wsfr	mem
wr	mem	[wsaddr]	meml	wr	meml	wsfr	meml
wr	mem	wsfr	wr'	wr	meml	mem	wr'
wr	mem	wsfr	wsaddr	wr	meml	mem	wsaddr
wr	mem	wsfr	wsfr	wr	meml	mem	wsfr
wr	mem	wsfr	mem	wr	meml	mem	mem
wr	mem	wsfr	meml	wr	meml	mem	meml
wr	mem	mem	wr'	wr	meml	meml	wr'
wr	mem	mem	wsaddr	wr	meml	meml	wsaddr
wr	mem	mem	wsfr	wr	meml	meml	wsfr
wr	mem	mem	mem	wr	meml	meml	mem
wr	mem	mem	meml	wr	meml	meml	meml
wr	mem	meml	wr'	wr	meml	PSW	wr'
wr	mem	meml	wsaddr	wr	meml	PSW	wsaddr
wr	mem	meml	wsfr	wr	meml	PSW	wsfr
wr	mem	meml	mem	wr	meml	PSW	mem
wr	mem	meml	meml	wr	meml	PSW	meml
wr	mem	PSW	wr'	wr	meml	SP	wr'
wr	mem	PSW	wsaddr	wr	meml	SP	wsaddr
wr	mem	PSW	wsfr	wr	meml	SP	wsfr
wr	mem	PSW	mem	wr	meml	SP	mem
wr	mem	PSW	meml	wr	meml	SP	meml
wr	mem	SP	wr'				
wr	mem	SP	wsaddr				
wr	mem	SP	wsfr				
wr	mem	SP	mem				
wr	mem	SP	meml				
wr	meml	wr'	wr'				
wr	meml	wr'	wsaddr				
wr	meml	wr'	wsfr				
wr	meml	wr'	mem				
wr	meml	wr'	meml				

A-2. Condition Expressions

Relational operator expression	Description format	Function
Equal	$\alpha == \beta$ [(γ)]	True if $\alpha = \beta$; False if $\alpha \neq \beta$.
Not Equal	$\alpha != \beta$ [(γ)]	True if $\alpha \neq \beta$; False if $\alpha = \beta$.
Less Than	$\alpha < \beta$ [(γ)]	True if $\alpha < \beta$; False if $\alpha >= \beta$.
Greater Than	$\alpha > \beta$ [(γ)]	True if $\alpha > \beta$; False if $\alpha <= \beta$.
Greater Than/Equal	$\alpha >= \beta$ [(γ)]	True if $\alpha >= \beta$; False if $\alpha < \beta$.
Less Than/Equal	$\alpha <= \beta$ [(γ)]	True if $\alpha <= \beta$; False if $\alpha > \beta$.

Bit condition expression	Description format	Function
Positive logic (bit)	bit symbol	True if specified bit is 1; otherwise, False.
Negative logic (bit)	!bit symbol	True if specified bit is 0; otherwise, False.

Logical operator expression	Description format	Function
AND	expression 1 && expression 2	True if both expression 1 and expression 2 are True.
OR	expression 1 || expression 2	True if either expression 1 or expression 2 is True.

A-3. Expression Statements

Substitution statement	Description format	Function
Substitute	$\alpha = \beta$	$\alpha \leftarrow \beta$
Substitute with Register name specification	$\alpha = \beta\ (\gamma)$	$\gamma \leftarrow \beta\ \ \alpha \leftarrow \gamma$
Add & Substitute	$\alpha += \beta$	$\alpha \leftarrow \alpha + \beta$
Subtract & Substitute	$\alpha -= \beta$	$\alpha \leftarrow \alpha - \beta$
Multiply & Substitute	$\alpha *= \beta$	$\alpha \leftarrow \alpha \times \beta$
Divide & Substitute	$\alpha /= \beta$	$\alpha \leftarrow \alpha \div = \beta$
AND & Substitute	$\alpha \&= \beta$	$\alpha \leftarrow \alpha \cap = \beta$
OR & Substitute	$\alpha \mid = \beta$	$\alpha \leftarrow \alpha \cup = \beta$
XOR & Substitute	$\alpha \hat{} = \beta$	$\alpha \leftarrow \alpha \hat{} \beta$
Shift Right & Substitute	$\alpha >>= \beta$	$(CY \leftarrow \alpha_0, \alpha_{n-1} \leftarrow \alpha_n, \alpha_{max} \leftarrow 0)$ $\times \beta$ times
Shift Left & Substitute	$\alpha <<= \beta$	$(CY \leftarrow \alpha_{max}, \alpha_{n-1} \leftarrow \alpha_n, \alpha_0 \leftarrow 0)$ $\times \beta$ times

Counter statement	Description format	Function
Increment	$\alpha ++$	$\alpha \leftarrow \alpha + 1$
Decrement	$\alpha --$	$\alpha \leftarrow \alpha - 1$

Exchange statement	Description format	Function
Exchange	$\alpha <-> \beta$	$\alpha \leftarrow \alpha <-> \beta$

A-4. Directives

Directive	Description format
#define	**#define** identifier character string
#ifdef	**#ifdef** identifier text 1 **#else** text 2 **#endif**
#include	**#include** "filename"
#defcallt	**#defcallt** label of CALLT table CALL label : **#endcallt**

APPENDIX B. LIST OF OPERANDS

B-1. Condition Expressions <78K/0/I/II/III>

α (alpha), β (beta), and registers that can be described in relational operator expressions are as shown below.

α	β	Register	0	I	II	III	α	β	Register	0	I	II	III
A	#byte	–	○	○	○	○	!addr16	r1	A	×	×	×	○
r	#byte	A	○	○	○	×	PSWH	r1	A	×	×	×	○
r1	#byte	A	×	×	×	○	PSWL	r1	A	×	×	×	○
[DE]	#byte	A	○	×	×	×	A	saddr	–	○	○	○	○
[HL]	#byte	A	○	○	×	×	r	saddr	A	○	○	○	×
[HL+byte]	#byte	A	○	×	×	×	r1	saddr	A	×	×	×	○
[HL + B]	#byte	A	○	×	×	×	saddr	saddr	–	×	×	○	○
[HL + C]	#byte	A	○	×	×	×	saddr	saddr	A	○	○	×	×
[r3]	#byte	A	×	○	×	×	[DE]	saddr	A	○	×	×	×
word[r1]	#byte	A	×	○	×	×	[HL]	saddr	A	○	○	×	×
sfr	#byte	A	○	×	×	×	[HL+byte]	saddr	A	○	×	×	×
mem	#byte	A	×	×	○	○	[HL + B]	saddr	A	○	×	×	×
&mem	#byte	A	×	×	○	×	[HL + C]	saddr	A	○	×	×	×
[saddrp]	#byte	A	×	×	×	○	[r3]	saddr	A	×	○	×	×
!addr16	#byte	A	○	×	○	○	word[r1]	saddr	A	×	○	×	×
&!addr16	#byte	A	×	×	○	×	sfr	saddr	A	○	○	○	○
PSW	#byte	A	○	○	○	×	mem	saddr	A	×	×	○	○
PSWH	#byte	A	×	×	×	○	&mem	saddr	A	×	×	○	×
PSWL	#byte	A	×	×	×	○	[saddrp]	saddr	A	×	×	×	○
saddr	#byte	–	○	○	○	○	!addr16	saddr	A	○	×	○	○
sfr	#byte	–	×	○	○	○	&!addr16	saddr	A	×	×	○	×
#byte	#byte	sfr	×	○	○	○	PSW	saddr	A	○	○	○	×
r	A	–	○	×	×	×	PSWH	saddr	A	×	×	×	○
#byte	A	r	○	×	×	×	PSWL	saddr	A	×	×	×	○
mem	A	–	×	×	×	○	A	sfr	–	×	○	○	○
r	r	–	×	○	○	×	r	sfr	A	×	○	○	×
#byte	r	r	○	○	○	×	r1	sfr	A	×	×	×	○
A	r	–	○	×	×	×	[HL]	sfr	A	×	○	×	×
saddr	r	A	×	○	○	×	[r3]	sfr	A	×	○	×	×
sfr	r	A	×	○	○	×	word[r1]	sfr	A	×	○	×	×
mem	r	A	×	×	○	×	saddr	sfr	A	×	○	○	○
&mem	r	A	×	×	○	×	sfr	sfr	A	×	○	○	○
!addr16	r	A	×	×	○	×	mem	sfr	A	×	×	○	○
&!addr16	r	A	×	×	○	×	&mem	sfr	A	×	×	○	×
[HL]	r	A	×	○	×	×	[saddrp]	sfr	A	×	×	×	○
[r3]	r	A	×	○	×	×	!addr16	sfr	A	×	×	○	○
word[r1]	r	A	×	○	×	×	&!addr16	sfr	A	×	×	○	×
PSW	r	A	×	○	○	×	PSW	sfr	A	×	○	○	×
r	r1	–	×	×	×	○	PSWH	sfr	A	×	×	×	○
#byte	r1	r1	×	×	×	○	PSWL	sfr	A	×	×	×	○
saddr	r1	A	×	×	×	○	A	[r4]	–	×	○	×	×
sfr	r1	A	×	×	×	○	r	[r4]	A	×	○	×	×
mem	r1	A	×	×	×	○	saddr	[r4]	A	×	○	×	×
[saddrp]	r1	A	×	×	×	○	sfr	[r4]	A	×	○	×	×

Note: r3, r4=00H to FFH (with uPD78112, r3, r4=40H to FFH)

B-1. Condition Expressions <78K/0/I/II/III> (contd)

α	β	Register	0	I	II	III	α	β	Register	0	I	II	III
[HL]	[r4]	A	×	○	×	×	[HL+byte]	[HL + B]	A	○	×	×	×
[r3]	[r4]	A	×	○	×	×	[HL + C]	[HL + B]	A	○	×	×	×
word[r1]	[r4]	A	×	○	×	×	PSW	[HL + B]	A	○	×	×	×
PSW	[r4]	A	×	○	×	×	A	[HL + C]	–	○	×	×	×
A	[HL]	–	○	○	×	×	r	[HL + C]	A	○	×	×	×
r	[HL]	A	○	○	×	×	saddr	[HL + C]	A	○	×	×	×
saddr	[HL]	A	○	○	×	×	sfr	[HL + C]	A	○	×	×	×
sfr	[HL]	A	○	○	×	×	[DE]	[HL + C]	A	○	×	×	×
[DE]	[HL]	A	○	×	×	×	[HL]	[HL + C]	A	○	×	×	×
!addr16	[HL]	A	○	×	×	×	!addr16	[HL + C]	A	○	×	×	×
[HL+byte]	[HL]	A	○	×	×	×	[HL+byte]	[HL + C]	A	○	×	×	×
[HL + B]	[HL]	A	○	×	×	×	[HL + B]	[HL + C]	A	○	×	×	×
[HL + C]	[HL]	A	○	×	×	×	PSW	[HL + C]	A	○	×	×	×
[r3]	[HL]	A	×	○	×	×	A	mem	–	×	×	○	○
word[r1]	[HL]	A	×	○	×	×	r	mem	A	×	×	○	×
PSW	[HL]	A	○	○	×	×	r1	mem	A	×	×	×	○
A	!addr16	–	○	×	×	×	saddr	mem	A	×	×	○	○
r	!addr16	A	○	×	×	×	sfr	mem	A	×	×	○	○
saddr	!addr16	A	○	×	×	×	mem	mem	A	×	×	○	○
sfr	!addr16	A	○	×	×	×	&mem	mem	A	×	×	○	×
[DE]	!addr16	A	○	×	×	×	[saddrp]	mem	A	×	×	×	○
[HL]	!addr16	A	○	×	×	×	!addr16	mem	A	×	×	○	○
!addr16	!addr16	A	○	×	×	×	&!addr16	mem	A	×	×	○	×
[HL+byte]	!addr16	A	○	×	×	×	PSW	mem	A	×	×	○	×
[HL + B]	!addr16	A	○	×	×	×	PSWH	mem	A	×	×	×	○
[HL + C]	!addr16	A	○	×	×	×	PSWL	mem	A	×	×	×	○
PSW	!addr16	A	○	×	×	×	A	&mem	–	×	×	○	×
A	[HL+byte]	–	○	×	×	×	r	&mem	A	×	×	○	×
r	[HL+byte]	A	○	×	×	×	saddr	&mem	A	×	×	○	×
saddr	[HL+byte]	A	○	×	×	×	sfr	&mem	A	×	×	○	×
sfr	[HL+byte]	A	○	×	×	×	mem	&mem	A	×	×	○	×
[DE]	[HL+byte]	A	○	×	×	×	&mem	&mem	A	×	×	○	×
[HL]	[HL+byte]	A	○	×	×	×	!addr16	&mem	A	×	×	○	×
!addr16	[HL+byte]	A	○	×	×	×	&!addr16	&mem	A	×	×	○	×
[HL + B]	[HL+byte]	A	○	×	×	×	PSW	&mem	A	×	×	○	×
[HL + C]	[HL+byte]	A	○	×	×	×	AX	#word	–	○	○	○	○
PSW	[HL+byte]	A	○	×	×	×	saddrp	#word	–	×	×	×	○
A	[HL + B]	–	○	×	×	×	sfrp	#word	–	×	×	×	○
r	[HL + B]	A	○	×	×	×	rp	#word	AX	○	○	○	×
saddr	[HL + B]	A	○	×	×	×	saddrp	#word	AX	○	○	○	×
sfr	[HL + B]	A	○	×	×	×	sfrp	#word	AX	○	○	○	×
[DE]	[HL + B]	A	○	×	×	×	!addr16	#word	AX	○	×	×	×
[HL]	[HL + B]	A	○	×	×	×	mem1	#word	AX	×	×	○	×
!addr16	[HL + B]	A	○	×	×	×	&mem1	#word	AX	×	×	○	×

B-1. Condition Expressions <78K/0/I/II/III> (contd)

α	β	Register	0	I	II	III	α	β	Register	0	I	II	III
#word	#word	sfrp	×	×	×	○	rp	saddrp	AX	×	○	○	×
AX	rp	–	×	○	○	×	rp1	saddrp	AX	×	×	×	○
rp	rp	AX	×	○	○	×	saddrp	saddrp	AX	×	○	○	×
saddrp	rp	AX	×	○	○	×	sfrp	saddrp	AX	×	○	○	○
sfrp	rp	AX	×	○	○	×	mem1	saddrp	AX	×	×	○	×
mem1	rp	AX	×	×	○	×	&mem1	saddrp	AX	×	×	○	×
&mem1	rp	AX	×	×	○	×	AX	sfrp	–	×	○	○	○
rp	rp1	–	×	×	×	○	rp	sfrp	AX	×	○	○	×
#word	rp1	rp1	×	×	×	○	rp1	sfrp	AX	×	×	×	○
saddrp	rp1	AX	×	×	×	○	saddrp	sfrp	AX	×	○	○	○
sfrp	rp1	AX	×	×	×	○	sfrp	sfrp	AX	×	○	○	○
AX	saddrp	–	×	○	○	○	mem1	sfrp	AX	×	×	○	×
saddrp	saddrp	–	×	×	×	○	&mem1	sfrp	AX	×	×	○	×

B-1. Condition Expressions <78KVI>

α	β	Register	α	β	Register
br	#byte	–	wr	#word	–
#byte	#byte	br	#word	#word	wr
br'	#byte	br	wr'	#word	wr
[wsaddr]	#byte	brl	[wsaddr]	#word	wr
bsaddr	#byte	–	PSW	#word	wr
bsfr	#byte	–	SP	#word	wr
#byte	#byte	bsfr	wsaddr	#word	–
br	br'	–	wsfr	#word	–
#byte	br'	br	#word	#word	wsfr
br'	br'	br	wr	wr'	–
mem	br'	br	#word	wr'	wr
bsaddr	br'	brl	wr'	wr'	wr
[wsaddr]	br'	brl	wsaddr	wr'	wr
bsfr	br'	brl	[wsaddr]	wr'	wr
meml	br'	brl	wsfr	wr'	wr
bsaddr	bsaddr'	–	mem	wr'	wr
brl	bsaddr	–	meml	wr'	wr
bsaddr	bsaddr	brl	PSW	wr'	wr
[wsaddr]	bsaddr	brl	SP	wr'	wr
bsfr	bsaddr	brl	wsaddr	wsaddr'	–
meml	bsaddr	brl	wr	wsaddr	–
bsaddr	brl	–	#word	wsaddr	wr
brl	bsfr	–	wr'	wsaddr	wr
bsaddr	bsfr	brl	wsaddr	wsaddr	wr
[wsaddr]	bsfr	brl	[wsaddr]	wsaddr	wr
bsfr	bsfr	brl	wsfr	wsaddr	wr
meml	bsfr	brl	mem	wsaddr	wr
br	mem	–	meml	wsaddr	wr
#byte	mem	br	PSW	wsaddr	wr
br'	mem	br	SP	wsaddr	wr
mem	mem	br	wsaddr	wr	–
bsaddr	mem	brl	wr	wsfr	–
[wsaddr]	mem	brl	#word	wsfr	wr
bsfr	mem	brl	wr'	wsfr	wr
meml	mem	brl	wsaddr	wsfr	wr
mem	br	–	[wsaddr]	wsfr	wr
brl	meml	–	wsfr	wsfr	wr
bsaddr	meml	brl	mem	wsfr	wr
[wsaddr]	meml	brl	meml	wsfr	wr
bsfr	meml	brl	PSW	wsfr	wr
meml	meml	brl	SP	wsfr	wr
meml	brl	–	wr	mem	–
mem	#byte	–	#word	mem	wr
meml	#byte	–	wr'	mem	wr

B-1. Condition Expressions <78KVI> (contd)

α	β	Register	α	β	Register
wsaddr	mem	wr	PSW	meml	wr
[wsaddr]	mem	wr	SP	meml	wr
wsfr	mem	wr	meml	wr	–
mem	mem	wr	mem	#word	–
meml	mem	wr	meml	#word	–
PSW	mem	wr	wrp	#dword	–
SP	mem	wr	#dword	#dword	wrp
mem	wr	–	dsaddr	#dword	wrp
wr	meml	wr	wrp	wrp'	-
#word	meml	wr	#dword	wrp'	wrp
wr'	meml	wr	dsaddr	wrp'	wrp
wsaddr	meml	wr	wrp	dsaddr	-
[wsaddr]	meml	wr	#dword	dsaddr	wrp
wsfr	meml	wr	dsaddr	dsaddr	wrp
mem	meml	wr	dsaddr	wrp	–
meml	meml	wr			

B-1. Condition Expressions <Bit symbols>

Bit symbols that can be described in bit condition expressions are as shown below.

Bit symbol	0	I	II	III	VI
CY	○	○	○	○	○
Z	○	○	○	○	○
saddr.bit	○	○	○	○	×
sfr.bit	○	○	○	○	×
A.bit	○	○	○	○	×
[HL].bit	○	×	×	×	×
X.bit	×	○	○	○	×
PSW.bit	○	○	○	×	×
PSWH.bit	×	×	×	○	×
PSWL.bit	×	×	×	○	×
br.bit3	×	×	×	×	○
br.br'	×	×	×	×	○
bsaddr.bit	×	×	×	×	○
bsaddr.br1	×	×	×	×	○
bsfr.bit3	×	×	×	×	○
bsfr.br1	×	×	×	×	○
mem.bit3	×	×	×	×	○
mem.br1	×	×	×	×	○
wr.bit4	×	×	×	×	○
wr.br	×	×	×	×	○

B2. Expression Statements <78K/0/I/II/III>

Expression		Operation	0	I	II	III
Substitute/Substitute with Register name specification	A = r	A←r	○	○	○	×
	A = r1	A←r1	×	×	×	○
	A = [DE]	A←(DE)	○	×	×	×
	A = [HL]	A←(HL)	○	○	×	×
	A = [HL+byte]	A←(HL+byte)	○	×	×	×
	A = [HL + B]	A←(HL + B)	○	×	×	×
	A = [HL + C]	A←(HL + C)	○	×	×	×
	A = [r3]	A←(FE00H + r3)	×	○	×	×
	A = word[r1]	A←(word + r1)	×	○	×	×
	A = saddr	A←(saddr)	○	○	○	○
	A = sfr	A←sfr	○	○	○	○
	A = mem	A←(mem)	×	×	○	○
	A = &mem	A←(&mem)	×	×	○	×
	A = [saddrp]	A←((saddrp))	×	×	×	○
	A = !addr16	A←(!addr16)	○	×	○	○
	A = &!addr16	A←(&!addr16)	×	×	○	×
	A = PSW	A←PSW	○	○	○	×
	A = PSWH	A←PSW_H	×	×	×	○
	A = PSWL	A←PSW_L	×	×	×	○
	r = #byte	r ←byte	○	○	○	×
	r = A	r ←A	○	×	×	×
	r = r	r←r	×	○	○	×
	r = r1	r←r1	×	×	×	○
	r = r (A)	r←r	○	×	×	×
	r = #byte (r1)	r ←byte	×	×	×	○
	r = [DE] (A)	r←(DE)	○	×	×	×
	r = [HL] (A)	r←(HL)	○	○	×	×
	r = [HL+byte] (A)	r←(HL +byte)	○	×	×	×
	r = [HL + B] (A)	r←(HL + B)	○	×	×	×
	r = [HL + C] (A)	r←(HL + C)	○	×	×	×
	r = [r3] (A)	r←(FE00H + r3)	×	○	×	×
	r = word[r1] (A)	r←(word + r1)	×	○	×	×
	r = saddr (A)	r←(saddr)	○	○	○	○
	r = sfr (A)	r←sfr	○	○	○	○
	r = mem (A)	r←(mem)	×	×	○	○
	r = &mem (A)	r←(&mem)	×	×	○	×
	r = [saddrp] (A)	r←((saddrp))	×	×	×	○
	r = !addr16 (A)	r←(!addr16)	○	×	○	○
	r = &!addr16 (A)	r←(&!addr16)	×	×	○	×
	r = PSW (A)	r←PSW	○	○	○	×
	A = PSWH	r←PSW_H	×	×	×	○
	A = PSWL	r←PSW_L	×	×	×	○
	r = #byte	r1←byte	○	○	○	○
	saddr = #byte	(saddr)←byte	○	○	○	○

Expression	Operation	0	I	II	III
Substitute/Substitute with Register name specification					
saddr = A	(saddr)←A	○	○	○	○
saddr = saddr	(saddr)←(saddr)	×	×	○	○
saddr = r (A)	(saddr)←r	○	○	○	○
saddr = r1 (A)	(saddr)←r1	×	×	×	○
saddr = [DE] (A)	(saddr)←(DE)	○	×	×	×
saddr = [HL] (A)	(saddr)←(HL)	○	○	×	×
saddr = [HL+byte] (A)	(saddr)←(HL+byte)	○	×	×	×
saddr = [HL + B] (A)	(saddr)←(HL + B)	○	×	×	×
saddr = [HL + C] (A)	A←(HL + C)	○	×	×	×
saddr = [r3] (A)	A←(FE00H + r3)	×	○	×	×
saddr = word[r1] (A)	A←(word+r1)	×	○	×	×
saddr = saddr (A)	(saddr)←(saddr)	○	○	×	×
saddr = sfr (A)	(saddr)←sfr	○	○	○	○
saddr = mem (A)	(saddr)←(mem)	×	×	○	○
saddr = &mem (A)	(saddr)←(&mem)	×	×	○	×
saddr = [saddrp] (A)	(saddr)←((saddrp))	×	×	×	○
saddr = !addr16 (A)	(saddr)←(!addr16)	○	×	○	×
saddr = &!addr16 (A)	(saddr)←(&!addr16)	×	×	○	×
saddr = PSW (A)	(saddr)←PSW	○	○	○	×
saddr = PSWH (A)	(saddr)←PSW_H	×	×	×	○
saddr = PSWL (A)	(saddr)←PSW_L	×	×	×	○
sfr = #byte	sfr←byte	○	○	○	○
sfr = A	sfr←A	○	○	○	○
sfr = r (A)	sfr←r	○	○	○	×
sfr = r1 (A)	sfr←r1	×	×	×	○
sfr = [DE] (A)	sfr←(DE)	○	×	×	×
sfr = [HL] (A)	sfr←(HL)	○	○	×	×
sfr = [HL+byte] (A)	sfr←(HL+byte)	○	×	×	×
sfr = [HL + B] (A)	sfr←(HL + B)	○	×	×	×
sfr = [HL + C] (A)	sfr←(HL + C)	○	×	×	×
sfr = [r3] (A)	sfr←(FE00H + r3)	×	○	×	×
sfr = word[r1] (A)	sfr←(word+r1)	×	○	×	×
sfr = saddr (A)	sfr←(saddr)	○	○	○	○
sfr = sfr (A)	sfr←sfr	○	○	○	○
sfr = mem (A)	sfr←(mem)	×	×	○	○
sfr = &mem (A)	sfr←(&mem)	×	×	○	×
sfr = [saddrp] (A)	sfr←((saddrp))	×	×	×	○
sfr = !addr16 (A)	sfr←(!addr16)	○	×	○	○
sfr = &!addr16 (A)	sfr←(&!addr16)	×	×	○	×
sfr = PSW (A)	sfr←PSW	○	○	○	×
sfr = PSWH (A)	sfr←PSW_H	×	×	×	○
sfr = PSWL (A)	sfr←PSW_L	×	×	×	○
[DE] = A	(DE)←A	○	×	×	×
[DE] = #byte (A)	(DE)←byte	○	×	×	×

Expression			Operation	0	I	II	III
Substitute/Substitute with Register name specification	[DE]	= r (A)	(HL)←r	○	×	×	×
	[DE]	= [HL] (A)	(HL)←(HL)	○	×	×	×
	[DE]	= [HL+byte] (A)	(HL)←(HL+byte)	○	×	×	×
	[DE]	= [HL + B] (A)	(HL)←(HL + B)	○	×	×	×
	[DE]	= [HL + C] (A)	(HL)←(HL + C)	○	×	×	×
	[DE]	= saddr (A)	(HL)←(saddr)	○	×	×	×
	[DE]	= sfr (A)	(HL)←sfr	○	×	×	×
	[DE]	= !addr16 (A)	(HL)←(!addr16)	○	×	×	×
	[DE]	= PSW (A)	(DE)←PSW	○	×	×	×
	[HL]	= A	(HL)←A	○	○	×	×
	[HL]	= #byte(A)	(HL)←byte	○	×	×	×
	[HL]	= r (A)	(HL)←r	○	○	×	×
	[HL]	= [DE] (A)	(HL)←(DE)	○	×	×	×
	[HL]	= [HL+byte] (A)	(HL)←(HL+byte)	○	×	×	×
	[HL]	= [HL + B] (A)	(HL)←(HL + B)	○	×	×	×
	[HL]	= [HL + C] (A)	(HL)←(HL + C)	○	×	×	×
	[HL]	= [r3] (A)	(HL)←(FE00H + r3)	×	○	×	×
	[HL]	= word[r1] (A)	(HL)←(word+r1)	×	○	×	×
	[HL]	= saddr (A)	(HL)←(saddr)	○	×	×	×
	[HL]	= sfr (A)	(HL)←sfr	○	○	×	×
	[HL]	= !addr16 (A)	(HL)←(!addr16)	○	×	×	×
	[HL]	= PSW (A)	(HL)←PSW	○	○	×	×
	[HL+byte]	= A	(HL+byte)←A	○	×	×	×
	[HL+byte]	= #byte (A)	(HL+byte)←byte	○	×	×	×
	[HL+byte]	= r (A)	(HL+byte)←r	○	×	×	×
	[HL+byte]	= [DE] (A)	(HL+byte)←(DE)	○	×	×	×
	[HL+byte]	= [HL] (A)	(HL+byte)←(HL)	○	×	×	×
	[HL+byte]	= [HL+byte] (A)	(HL+byte)←(HL+byte)	○	×	×	×
	[HL+byte]	= [HL + B] (A)	(HL+byte)←(HL + B)	○	×	×	×
	[HL+byte]	= [HL + C] (A)	(HL+byte)←(HL + C)	○	×	×	×
	[HL+byte]	= saddr (A)	(HL+byte)←(saddr)	○	×	×	×
	[HL+byte]	= sfr (A)	(HL+byte)←sfr	○	×	×	×
	[HL+byte]	= !addr16 (A)	(HL+byte)←(!addr16)	○	×	×	×
	[HL+byte]	= PSW (A)	(HL+byte)←PSW	○	×	×	×
	[HL + B]	= A	(HL + B)←A	○	×	×	×
	[HL + B]	= #byte (A)	(HL + B)←byte	○	×	×	×
	[HL + B]	= r (A)	(HL + B)←r	○	×	×	×
	[HL + B]	= [DE] (A)	(HL + B)←(DE)	○	×	×	×
	[HL + B]	= [HL] (A)	(HL + B)←(HL)	○	×	×	×
	[HL + B]	= [HL+byte] (A)	(HL + B)←(HL+byte)	○	×	×	×
	[HL + B]	= [HL + C] (A)	(HL + B)←(HL + C)	○	×	×	×
	[HL + B]	= saddr (A)	(HL + B)←(saddr)	○	×	×	×
	[HL + B]	= sfr (A)	(HL + B)←sfr	○	×	×	×
	[HL + B]	= !addr16 (A)	(HL + B)←(!addr16)	○	×	×	×

Expression			Operation	0	I	II	III
Substitute/Substitute with Register name specification	[HL + B]	= PSW (A)	(HL + B)←PSW	○	×	×	×
	[HL + C]	= A	(HL + C)←A	○	×	×	×
	[HL + C]	= #byte (A)	(HL + C)←byte	○	×	×	×
	[HL + C]	= r (A)	(HL + C)← r	○	×	×	×
	[HL + C]	= [DE] (A)	(HL + C)←(DE)	○	×	×	×
	[HL + C]	= [HL] (A)	(HL + C)←(HL)	○	×	×	×
	[HL + C]	= [HL+byte] (A)	(HL + C)←(HL+byte)	○	×	×	×
	[HL + C]	= [HL + B] (A)	(HL + C)←(HL + B)	○	×	×	×
	[HL + C]	= saddr (A)	(HL + C)←(saddr)	○	×	×	×
	[HL + C]	= sfr (A)	(HL + C)←sfr	○	×	×	×
	[HL + C]	= !addr16 (A)	(HL + C)←(!addr16)	○	×	×	×
	[HL + C]	= PSW (A)	(HL + C)←PSW	○	×	×	×
	[r3]	= A	(FE00 + r3)←A	×	○	×	×
	[r3]	= #byte (A)	(FE00 + r3)←byte	×	○	×	×
	[r3]	= r (A)	(FE00 + r3)←r	×	○	×	×
	[r3]	= [HL] (A)	(FE00 + r3)←(HL)	×	○	×	×
	[r3]	= [r3] (A)	(FE00 + r3)←(FE00H + r3)	×	○	×	×
	[r3]	= word[r1] (A)	(FE00 + r3)←(word+r1)	×	○	×	×
	[r3]	= saddr (A)	(FE00 + r3)←(saddr)	×	○	×	×
	[r3]	= sfr (A)	(FE00 + r3)←sfr	×	○	×	×
	[r3]	= PSW (A)	(FE00 + r3)←PSW	×	○	×	×
	word[r1]	= A	(word + r1)←A	×	○	×	×
	word[r1]	= #byte (A)	(word + r1)←byte	×	○	×	×
	word[r1]	= r (A)	(word+r1)←r	×	○	×	×
	word[r1]	= [HL] (A)	(word+r1)←(HL)	×	○	×	×
	word[r1]	= [r3] (A)	(word+r1)←(FE00 + r3)	×	○	×	×
	word[r1]	= word[r1] (A)	(word+r1)←(word+r1)	×	○	×	×
	word[r1]	= saddr (A)	(word+r1)←(saddr)	×	○	×	×
	word[r1]	= sfr (A)	(word+r1)←sfr	×	○	×	×
	word[r1]	= PSW (A)	(word+r1)←PSW	×	○	×	×
	mem	= A	(mem)←A	×	×	○	○
	mem	= #byte (A)	(mem)←byte	×	×	○	○
	mem	= r (A)	(mem)←r	×	×	○	×
	mem	= r1 (A)	(mem)←r1	×	×	×	○
	mem	= saddr (A)	(mem)←(saddr)	×	×	○	○
	mem	= sfr (A)	(mem)←sfr	×	×	○	○
	mem	= mem (A)	(mem)←(mem)	×	×	○	○
	mem	= &mem (A)	(mem)←(&mem)	×	×	○	×
	mem	= [saddrp] (A)	(mem)←((saddrp))	×	×	×	○
	mem	= !addr16 (A)	(mem)←(!addr16)	×	×	○	○
	mem	= &!addr16 (A)	(mem)←(&!addr16)	×	×	○	×
	mem	= PSW (A)	(mem)←PSW	×	×	○	×
	mem	= PSWH (A)	(mem)←PSW_H	×	×	×	○
	mem	= PSWL (A)	(mem)←PSW_L	×	×	×	○

Expression			Operation	0	I	II	III
Substitute/Substitute with Register name specification	&mem = A		(&mem)←A	×	×	○	×
	&mem = #byte (A)		(&mem)←byte	×	×	○	×
	&mem = r (A)		(&mem)←r	×	×	○	×
	&mem = saddr (A)		(&mem)←(saddr)	×	×	○	×
	&mem = sfr (A)		(&mem)←sfr	×	×	○	×
	&mem = mem (A)		(&mem)←(mem)	×	×	○	×
	&mem = &mem (A)		(&mem)←(&mem)	×	×	○	×
	&mem = !addr16 (A)		(&mem)←!addr16 (A)	×	×	○	×
	&mem = &!addr16 (A)		(&mem)←(&!addr16)	×	×	○	×
	&mem = PSW (A)		(&mem)←PSW	×	×	○	×
	[saddrp] = A		((saddrp))←A	×	×	×	○
	[saddrp] = #byte (A)		((saddrp))←byte	×	×	×	○
	[saddrp] = r1 (A)		((saddrp))←r1	×	×	×	○
	[saddrp] = saddr (A)		((saddrp))←(saddr)	×	×	×	○
	[saddrp] = sfr (A)		((saddrp))←sfr	×	×	×	○
	[saddrp] = mem (A)		((saddrp))←(mem)	×	×	×	○
	[saddrp] = [saddrp] (A)		((saddrp))←((saddrp))	×	×	×	○
	[saddrp] = !addr16 (A)		((saddrp))←(!addr16)	×	×	×	○
	[saddrp] = PSWH (A)		((saddrp))←PSW_H	×	×	×	○
	[saddrp] = PSWL (A)		((saddrp))←PSW_L	×	×	×	○
	!addr16 = A		(!addr16)←A	○	×	○	○
	!addr16 = AX		(!addr16)←AX	○	×	×	×
	!addr16 = #byte (A)		(!addr16)←byte	○	×	○	○
	!addr16 = r (A)		(!addr16)←r	○	×	○	×
	!addr16 = r1 (A)		(!addr16)←r1	×	×	×	○
	!addr16 = [DE] (A)		(!addr16)←(DE)	○	×	×	×
	!addr16 = [HL] (A)		(!addr16)←(HL)	○	×	×	×
	!addr16 = [HL+byte] (A)		(!addr16)←(HL+byte)	○	×	×	×
	!addr16 = [HL + B] (A)		(!addr16)←(HL + B)	○	×	×	×
	!addr16 = [HL + C] (A)		(!addr16)←(HL + C)	○	×	×	×
	!addr16 = saddr (A)		(!addr16)←(saddr)	○	×	○	○
	!addr16 = sfr (A)		(!addr16)←sfr	○	×	○	○
	!addr16 = mem (A)		(!addr16)←(mem)	×	×	○	○
	!addr16 = &mem (A)		(!addr16)←(&mem)	×	×	○	×
	!addr16 = [saddrp] (A)		(!addr16)←((saddrp))	×	×	×	○
	!addr16 = !addr16 (A)		(!addr16)←(!addr16)	○	×	○	○
	!addr16 = &!addr16 (A)		(!addr16)←(&!addr16)	×	×	○	×
	!addr16 = PSW (A)		(!addr16)←PSW	○	×	○	×
	!addr16 = PSWH (A)		(!addr16)←PSW_H	×	×	×	○
	!addr16 = PSWL (A)		(!addr16)←PSW_L	×	×	×	○
	!addr16 = #word (AX)		(!addr16)←word	○	×	×	×
	!addr16 = rp (AX)		(!addr16)←rp	○	×	×	×
	!addr16 = saddrp (AX)		(!addr16)←(saddrp)	○	×	×	×
	!addr16 = sfrp (AX)		(!addr16)←sfrp	○	×	×	×

B2. Expression Statements <78K/0/I/II/III> (contd)

Expression				Operation	0	I	II	III
Substitute/Substitute with Register name specification	!addr16	=	!addr16 (AX)	(!addr16)←(!addr16)	○	×	×	×
	&!addr16	=	A	(&!addr16)←A	×	×	○	×
	&!addr16	=	#byte (A)	(&!addr16)←byte	×	×	○	×
	&!addr16	=	r (A)	(&!addr16)←r	×	×	○	×
	&!addr16	=	saddr (A)	(&!addr16)←(saddr)	×	×	○	×
	&!addr16	=	sfr (A)	(&!addr16)←sfr	×	×	○	×
	&!addr16	=	mem (A)	(&!addr16)←(mem)	×	×	○	×
	&!addr16	=	&mem (A)	(&!addr16)←(&mem)	×	×	○	×
	&!addr16	=	!addr16 (A)	(&!addr16)←(!addr16)	×	×	○	×
	&!addr16	=	&!addr16 (A)	(&!addr16)←(&!addr16)	×	×	○	×
	&!addr16	=	PSW (A)	(&!addr16)←PSW	×	×	○	×
	PSW	=	#byte	PSW←byte	○	○	○	×
	PSW	=	A	PSW←A	○	○	○	×
	PSW	=	r (A)	PSW←r	○	○	○	×
	PSW	=	[DE] (A)	PSW←(DE)	○	×	×	×
	PSW	=	[HL] (A)	PSW←(HL)	○	○	×	×
	PSW	=	[HL+byte] (A)	PSW←(HL+byte)	○	×	×	×
	PSW	=	[HL+B] (A)	PSW←(HL+B)	○	×	×	×
	PSW	=	[HL+C] (A)	PSW←(HL+C)	○	×	×	×
	PSW	=	[r3] (A)	PSW←(FE00H+r3)	×	○	×	×
	PSW	=	word[r1] (A)	PSW←(word+r1)	×	○	×	×
	PSW	=	saddr (A)	PSW←(saddr)	○	○	○	×
	PSW	=	sfr (A)	PSW←sfr	○	○	○	×
	PSW	=	mem (A)	PSW←(mem)	×	×	○	×
	PSW	=	&mem (A)	PSW←(&mem)	×	×	○	×
	PSW	=	!addr16 (A)	PSW←(!addr16)	○	×	○	×
	PSW	=	&!addr16 (A)	PSW←(&!addr16)	×	×	○	×
	PSW	=	PSW (A)	PSW←PSW	○	○	○	×
	PSWH	=	#byte	PSW_H←byte	×	×	×	○
	PSWH	=	A	PSW_H←A	×	×	×	○
	PSWH	=	r1 (A)	PSW_H←r1	×	×	×	○
	PSWH	=	saddr (A)	PSW_H←(saddr)	×	×	×	○
	PSWH	=	sfr (A)	PSW_H←sfr	×	×	×	○
	PSWH	=	mem (A)	PSW_H←(mem)	×	×	×	○
	PSWH	=	[saddrp] (A)	PSW_H←((saddrp))	×	×	×	○
	PSWH	=	!addr16 (A)	PSW_H←(!addr16)	×	×	×	○
	PSWH	=	PSWH (A)	PSW_H←PSW_H	×	×	×	○
	PSWH	=	PSWL (A)	PSW_H←PSW_L	×	×	×	○
	PSWL	=	#byte	PSW_L←byte	×	×	×	○
	PSWL	=	A	PSW_L←A	×	×	×	○
	PSWL	=	r1 (A)	PSW_L←r1	×	×	×	○
	PSWL	=	saddr (A)	PSW_L←(saddr)	×	×	×	○
	PSWL	=	sfr (A)	PSW_L←sfr	×	×	×	○
	PSWL	=	mem (A)	PSW_L←(mem)	×	×	×	○

Expression		Operation	0	I	II	III
Substitute/Substitute with Register name specification	PSWL = [saddrp] (A)	$PSW_L \leftarrow ((saddrp))$	×	×	×	○
	PSWL = !addr16 (A)	$PSW_L \leftarrow (!addr16)$	×	×	×	○
	PSWL = PSWH (A)	$PSW_L \leftarrow PSW_H$	×	×	×	○
	PSWL = PSWL (A)	$PSW_L \leftarrow PSW_L$	×	×	×	○
	rp = #word	rp←word	○	○	○	×
	rp = AX	rp←AX	○	×	×	×
	rp = rp	rp←rp	×	○	○	×
	rp = rp1	rp←rp1	×	×	×	○
	rp = #word (AX)	rp←word	×	×	×	×
	rp = rp (AX)	rp←rp	○	×	×	×
	rp = saddrp (AX)	rp←(saddrp)	○	○	○	○
	rp = sfrp (AX)	rp←sfrp	○	○	○	○
	rp = mem1 (AX)	rp←(mem1)	×	×	○	×
	rp = &mem1 (AX)	rp←(&mem1)	×	×	○	×
	rp = !addr16 (AX)	rp←(!addr16)	○	×	×	×
	rp1 = #word	rp1←word	×	×	×	○
	saddrp = #word	(saddrp)←word	○	○	○	○
	saddrp = AX	(saddrp)←AX	○	○	○	○
	saddrp = saddrp	(saddrp)←(saddrp)	×	×	×	○
	saddrp = rp (AX)	(saddrp)←rp	×	○	○	×
	saddrp = rp1 (AX)	(saddrp)←rp1	×	×	×	○
	saddrp = saddrp (AX)	(saddrp)←(saddrp)	○	○	○	○
	saddrp = sfrp (AX)	(saddrp)←sfrp	○	○	○	○
	saddrp = mem1 (AX)	(saddrp)←(mem1)	×	×	○	×
	saddrp = &mem1 (AX)	(saddrp)←(&mem1)	×	×	○	×
	saddrp = !addr16 (AX)	(saddrp)←(!addr16)	○	×	×	×
	sfrp = #word	sfrp←word	○	○	○	○
	sfrp = AX	sfrp←AX	○	○	○	○
	sfrp = rp (AX)	sfrp←rp	×	○	○	×
	sfrp = rp1 (AX)	sfrp←rp1	×	×	×	○
	sfrp = saddrp (AX)	sfrp←(saddrp)	○	○	○	○
	sfrp = sfrp (AX)	sfrp←sfrp	○	○	○	○
	sfrp = mem1 (AX)	sfrp←(mem1)	×	×	○	×
	sfrp = &mem1 (AX)	sfrp←(&mem1)	×	×	○	×
	sfrp = !addr16 (AX)	sfrp←(!addr16)	○	×	×	×
	AX = saddrp	AX←(saddrp)	○	○	○	○
	AX = rp	AX←rp	○	○	×	×
	AX = sfrp	AX←sfrp	○	○	○	○
	AX = mem1	AX←(mem1)	×	×	○	×
	AX = &mem1	AX←(&mem1)	×	×	○	×
	AX = !addr16	AX←(!addr16)	○	×	×	×
	mem1 = AX	(mem1)←AX	×	×	○	×
	mem1 = #word (AX)	(mem1)←word	×	×	○	×
	mem1 = rp (AX)	(mem1)←(rp)	×	×	○	×

B2. Expression Statements <78K/0/I/II/III> (contd)

	Expression	Operation	0	I	II	III
Substitute/Substitute with Register name specification	mem1 = saddrp (AX)	(mem1)←(saddrp)	×	×	○	×
	mem1 = sfrp (AX)	(mem1)←sfrp	×	×	○	×
	mem1 = mem1 (AX)	(mem1)←(mem1)	×	×	○	×
	mem1 = &mem1 (AX)	(mem1)←(&mem1)	×	×	○	×
	&mem1 = AX	(&mem1)←AX	×	×	○	×
	&mem1 = #word (AX)	(&mem1)←word	×	×	○	×
	&mem1 = rp (AX)	(&mem1)←(rp)	×	×	○	×
	&mem1 = saddrp (AX)	(&mem1)←(saddrp)	×	×	○	×
	&mem1 = sfrp (AX)	(&mem1)←sfrp	×	×	○	×
	&mem1 = mem1 (AX)	(&mem1)←(mem1)	×	×	○	×
	&mem1 = &mem1 (AX)	(&mem1)←(&mem1)	×	×	○	×
	CY = saddr.bit	CY←(saddr.bit)	○	○	○	○
	CY = sfr.bit	CY←sfr.bit	○	○	○	○
	CY = X.bit	CY←X.bit	○	○	○	○
	CY = A.bit	CY←A.bit	×	○	○	○
	CY = PSW.bit	CY←PSW.bit	○	○	○	×
	CY = PSWH.bit	CY←PSW_H.bit	×	×	×	○
	CY = PSWL.bit	CY←PSW_L.bit	×	×	×	○
	CY = [HL].bit	CY←(HL).bit	○	×	×	×
	saddr.bit = CY	(saddr.bit)←CY	○	○	○	○
	saddr.bit = saddr.bit (CY)	(saddr.bit)←(saddr.bit)	○	○	○	○
	saddr.bit = sfr.bit (CY)	(saddr.bit)←sfr.bit	○	○	○	○
	saddr.bit = A.bit (CY)	(saddr.bit)←A.bit	○	○	○	○
	saddr.bit = X.bit (CY)	(saddr.bit)←X.bit	×	○	○	○
	saddr.bit = PSW.bit (CY)	(saddr.bit)←PSW.bit	○	○	○	×
	saddr.bit = PSWH.bit (CY)	(saddr.bit)←PSWH.bit	×	×	×	○
	saddr.bit = PSWL.bit (CY)	(saddr.bit)←PSWL.bit	×	×	×	○
	saddr.bit = [HL].bit (CY)	(saddr.bit)←(HL).bit	○	×	×	×
	sfr.bit = CY	sfr.bit←CY	○	○	○	○
	sfr.bit = saddr.bit (CY)	sfr.bit←(saddr.bit)	○	○	○	○
	sfr.bit = sfr.bit (CY)	sfr.bit←sfr.bit	○	○	○	○
	sfr.bit = A.bit (CY)	sfr.bit←A.bit	○	○	○	○
	sfr.bit = X.bit (CY)	sfr.bit←X.bit	×	○	○	○
	sfr.bit = PSW.bit (CY)	sfr.bit←PSW.bit	○	○	○	×
	sfr.bit = PSWH.bit (CY)	sfr.bit←PSW_H.bit	×	×	×	○
	sfr.bit = PSWL.bit (CY)	sfr.bit←PSW_L.bit	×	×	×	○
	sfr.bit = [HL].bit (CY)	sfr.bit←(HL).bit	○	×	×	×
	A.bit = CY	A.bit←CY	○	○	○	○
	A.bit = saddr.bit (CY)	A.bit←(saddr.bit)	○	○	○	○
	A.bit = sfr.bit (CY)	A.bit←sfr.bit	○	○	○	○
	A.bit = A.bit (CY)	A.bit←A.bit	○	○	○	○
	A.bit = X.bit (CY)	A.bit←X.bit	×	○	○	○
	A.bit = PSW.bit (CY)	A.bit←PSW.bit	○	○	○	×
	A.bit = PSWH.bit (CY)	A.bit←PSW_H.bit	×	×	×	○

Expression		Operation	0	I	II	III
Substitute/Substitute with Register name specification	A.bit = PSWL.bit (CY)	A.bit←PSW_L.bit	×	×	×	○
	A.bit = [HL].bit (CY)	A.bit←(HL).bit	○	×	×	×
	X.bit = CY	X.bit←CY	×	○	○	○
	X.bit = saddr.bit (CY)	X.bit←(saddr.bit)	×	○	○	○
	X.bit = sfr.bit (CY)	X.bit←sfr.bit	×	○	○	○
	X.bit = A.bit (CY)	X.bit←A.bit	×	○	○	○
	X.bit = X.bit (CY)	X.bit←X.bit	×	○	○	○
	X.bit = PSW.bit (CY)	X.bit←PSW.bit	×	○	○	×
	X.bit = PSWH.bit (CY)	X.bit←PSW_H.bit	×	×	×	○
	X.bit = PSWL.bit (CY)	X.bit←PSW_L.bit	×	×	×	○
	PSW.bit = CY	PSW.bit←CY	○	○	○	×
	PSW.bit = saddr.bit (CY)	PSW.bit←(saddr.bit)	○	○	○	×
	PSW.bit = sfr.bit (CY)	PSW.bit←sfr.bit	○	○	○	×
	PSW.bit = A.bit (CY)	PSW.bit←A.bit	○	○	○	×
	PSW.bit = X.bit (CY)	PSW.bit←X.bit	×	○	○	×
	PSW.bit = PSW.bit (CY)	PSW.bit←PSW.bit	○	○	○	×
	PSW.bit = [HL].bit (CY)	PSW.bit←(HL).bit	○	×	×	×
	PSWH.bit = CY	PSW_H.bit←CY	×	×	×	○
	PSWH.bit = saddr.bit (CY)	PSW_H.bit←(saddr.bit)	×	×	×	○
	PSWH.bit = sfr.bit (CY)	PSW_H.bit←sfr.bit	×	×	×	○
	PSWH.bit = A.bit (CY)	PSW_H.bit←A.bit	×	×	×	○
	PSWH.bit = X.bit (CY)	PSW_H.bit←X.bit	×	×	×	○
	PSWH.bit = PSWH.bit (CY)	PSW_H.bit←PSW_H.bit	×	×	×	○
	PSWH.bit = PSWL.bit (CY)	PSW_H.bit←PSW_L.bit	×	×	×	○
	PSWL.bit = CY	PSW_L.bit←CY	×	×	×	○
	PSWL.bit = saddr.bit (CY)	PSW_L.bit←(saddr.bit)	×	×	×	○
	PSWL.bit = sfr.bit (CY)	PSW_L.bit←sfr.bit	×	×	×	○
	PSWL.bit = A.bit (CY)	PSW_L.bit←A.bit	×	×	×	○
	PSWL.bit = X.bit (CY)	PSW_L.bit←X.bit	×	×	×	○
	PSWL.bit = PSWH.bit (CY)	PSW_L.bit←PSW_H.bit	×	×	×	○
	PSWL.bit = PSWL.bit (CY)	PSW_L.bit←PSW_L.bit	×	×	×	○
	[HL].bit = CY	(HL).bit ←CY	○	×	×	×
	[HL].bit = saddr.bit (CY)	(HL).bit ←(saddr.bit)	○	×	×	×
	[HL].bit = sfr.bit (CY)	(HL).bit ←sfr.bit	○	×	×	×
	[HL].bit = A.bit (CY)	(HL).bit ←A.bit	○	×	×	×
	[HL].bit = PSW.bit (CY)	(HL).bit ←PSW.bit	○	×	×	×
	[HL].bit = [HL].bit (CY)	(HL).bit ←(HL).bit	○	×	×	×

B2. Expression Statements <78K/0/I/II/III> (contd)

	Expression	Operation	0	I	II	III
Add & Substitute (Note 1)	A += r	A, CY←A + r	○	×	×	×
	A += [HL]	A, CY←A + (HL)	○	○	×	×
	A += !addr16	A, CY←A + (!addr16)	○	×	×	×
	A += [HL + byte]	A, CY←A + (HL + byte)	○	×	×	×
	A += [HL + B]	A, CY←A + (HL + B)	○	×	×	×
	A += [HL + C]	A, CY←A + (HL + C)	○	×	×	×
	A += [r4]	A, CY←A + (FE00H + r4)	×	○	×	×
	A += #byte	A, CY←A + byte	○	○	○	○
	A += saddr	A, CY←A + (saddr)	⊙	○	⊙	○
	A += sfr	A, CY←A + sfr	×	○	○	○
	A += mem	A, CY←A + (mem)	×	×	○	○
	A += &mem	A, CY←A + (&mem)	×	×	○	×
	saddr += #byte	(saddr), CY←(saddr) + byte	○	○	○	○
	saddr += saddr	(saddr), CY←(saddr) + (saddr)	×	×	○	○
	sfr += #byte	sfr, CY←sfr + byte	×	○	○	○
	r += A	r, CY←r + A	○	×	×	×
	r += r	r, CY←r + r	×	○	○	×
	r += r1	r, CY←r + r1	×	×	×	○
	mem += A	(mem), CY←(mem) + A	×	×	×	○
	AX += #word	AX, CY←AX + word	○	○	○	○
	AX += rp	AX, CY←AX + rp	×	○	○	×
	AX += saddrp	AX, CY←AX + (saddrp)	×	○	○	○
	AX += sfrp	AX, CY←AX + sfrp	×	○	○	○
	saddrp += #word	(saddrp), CY←(saddrp) + word	×	×	×	○
	saddrp += saddrp	(saddrp), CY←(saddrp) + (saddrp)	×	×	×	○
	sfrp += #word	sfrp, CY←sfrp + word	×	×	×	○
	rp += rp1	rp, CY←rp + rp1	×	×	×	○
Subtract & Substitute (Note 1)	A -= r	A, CY←A - r	○	×	×	×
	A -= [HL]	A, CY←A - (HL)	○	○	×	×
	A -= !addr16	A, CY←A - (!addr16)	○	×	×	×
	A -= [HL + byte]	A, CY←A - (HL + byte)	○	×	×	×
	A -= [HL + B]	A, CY←A - (HL + B)	○	×	×	×
	A -= [HL + C]	A, CY←A - (HL + C)	○	×	×	×
	A -= [r4]	A, CY←A - (FE00H + r4)	×	○	×	×
	A -= #byte	A, CY←A - byte	○	○	○	○
	A -= saddr	A, CY←A - (saddr)	○	○	○	○
	A -= sfr	A, CY←A - sfr	×	○	○	○
	A -= mem	A, CY←A - (mem)	×	×	○	○
	A -= &mem	A, CY←A - (&mem)	×	×	○	×
	saddr -= #byte	(saddr), CY←(saddr) - byte	○	○	○	○
	saddr -= saddr	(saddr), CY←(saddr) - (saddr)	×	×	○	○
	sfr -= #byte	sfr, CY←sfr - byte	×	○	○	○
	r -= A	r, CY←r - A	○	×	×	×

Note 1: If Add or Subtract with Carry is to be performed, describe the assembly language without change.

		Expression	Operation	0	I	II	III
Note 1	Subtract & Substitute	r -= r	r, CY ← r − r	×	○	○	×
		r -= rl	r, CY ← r − rl	×	×	×	○
		mem -= A	(mem), CY ← (mem) − A	×	×	×	○
		AX -= #word	AX, CY ← AX − word	○	○	○	○
		AX -= rp	AX, CY ← AX − rp	×	○	○	×
		AX -= saddrp	AX, CY ← AX − (saddrp)	×	○	○	○
		AX -= sfrp	AX, CY ← AX − sfrp	×	○	○	○
		saddrp -= #word	(saddrp), CY ← (saddrp) − word	×	×	×	○
		saddrp -= saddrp	(saddrp), CY ← (saddrp) − (saddrp)	×	×	×	○
		sfrp -= #word	sfrp, CY ← sfrp − word	×	×	×	○
		rp -= rpl	rp, CY ← rp − rpl	×	×	×	○
Note 2	Multiply & Substitute	AX *= X	AX ← A * X	○	×	×	×
		AX *= r	AX ← A * r	×	○	○	×
		AX *= rl	AX ← AX * rl	×	×	×	○
		AX *= rpl	AX (high 16 bits), rpl (low 16 bits) ← AX * rpl	×	×	×	○
	Divide & Substitute	AX /= C	AX(quo), C (rem) ← AX ÷ C	○	×	×	×
		AX /= r	AX(quo), r (rem) ← AX ÷ r	×	○	○	×
		AX /= rl	AX(quo), rl (rem) ← AX ÷ rl	×	×	×	○
		AX /= rpl	AXDE(商), rpl (rem) ← AXDE ÷ rpl	×	×	×	○

quo: Quotient; rem: Remainder

Note: 1. If Add or Subtract with Carry is to be performed, describe the assembly language without change.
2. With a signed Multiply instruction, describe the assembly language without change.

Expression			Operation	0	I	II	III
AND & Substitute	A	&= r	A←A∩r	○	×	×	×
	A	&= [HL]	A←A∩(HL)	○	○	×	×
	A	&= !addr16	A←A∩(!addr16)	○	×	×	×
	A	&= [HL+byte]	A←A∩(HL+byte)	○	×	×	×
	A	&= [HL+B]	A←A∩(HL+B)	○	×	×	×
	A	&= [HL+C]	A←A∩(HL+C)	○	×	×	×
	A	&= [r4]	A←A∩(FE00H+r4)	×	○	×	×
	A	&= #byte	A←A∩byte	○	○	○	○
	A	&= saddr	A←A∩(saddr)	○	○	○	○
	A	&= sfr	A←A∩sfr	×	○	○	○
	A	&= mem	A←A∩(mem)	×	○	○	○
	A	&= &mem	A←A∩(&mem)	×	×	○	×
	saddr	&= #byte	(saddr)←(saddr)∩byte	○	○	○	○
	saddr	&= saddr	(saddr)←(saddr)∩(saddr)	×	○	○	○
	sfr	&= #byte	sfr←sfr∩byte	×	×	○	○
	r	&= A	r←r∩A	○	×	×	×
	r	&= r	r←r∩r	×	×	○	×
	r	&= r1	r←r∩r1	×	×	×	○
	mem	&= A	(mem)←(mem)∩A	×	×	×	○
	CY	&= [HL].bit	CY←CY∩(HL).bit	○	×	×	×
	CY	&= saddr.bit	CY←CY∩(saddr.bit)	○	○	○	○
	CY	&= /saddr.bit	CY←CY∩$\overline{(saddr.bit)}$	×	○	○	○
	CY	&= sfr.bit	CY←CY∩sfr.bit	○	○	○	○
	CY	&= /sfr.bit	CY←CY∩$\overline{sfr.bit}$	×	○	○	○
	CY	&= A.bit	CY←CY∩A.bit	○	○	○	○
	CY	&= /A.bit	CY←CY∩$\overline{A.bit}$	×	○	○	○
	CY	&= X.bit	CY←CY∩X.bit	×	○	○	○
	CY	&= /X.bit	CY←CY∩$\overline{X.bit}$	×	○	○	○
	CY	&= PSW.bit	CY←CY∩PSW.bit	○	○	○	×
	CY	&= /PSW.bit	CY←CY∩$\overline{PSW.bit}$	×	○	○	×
	CY	&= PSWH.bit	CY←CY∩PSW_H.bit	×	×	×	○
	CY	&= /PSWH.bit	CY←CY∩$\overline{PSW_H.bit}$	×	×	×	○
	CY	&= PSWL.bit	CY←CY∩PSW_L.bit	×	×	×	○
	CY	&= /PSWL.bit	CY←CY∩$\overline{PSW_L.bit}$	×	×	×	○

Expression			Operation	0	I	II	III
OR & Substitute	A	|= r	A←A∪r	○	×	×	×
	A	|= [HL]	A←A∪(HL)	○	○	×	×
	A	|= !addr16	A←A∪(!addr16)	○	×	×	×
	A	|= [HL+byte]	A←A∪(HL+byte)	○	×	×	×
	A	|= [HL+B]	A←A∪(HL+B)	○	×	×	×
	A	|= [HL+C]	A←A∪(HL+C)	○	×	×	×
	A	|= [r4]	A←A∪(FE00H+r4)	×	○	×	×
	A	|= #byte	A←A∪byte	○	○	○	○
	A	|= saddr	A←A∪(saddr)	○	○	○	○
	A	|= sfr	A←A∪sfr	×	○	○	○
	A	|= mem	A←A∪(mem)	×	○	○	○
	A	|= &mem	A←A∪(&mem)	×	×	○	×
	saddr	|= #byte	(saddr)←(saddr)∪byte	○	○	○	○
	saddr	|= saddr	(saddr)←(saddr)∪(saddr)	×	○	○	○
	sfr	|= #byte	sfr←sfr∪byte	×	×	○	○
	r	|= A	r←r∪A	○	×	×	×
	r	|= r	r←r∪r	×	×	○	×
	r	|= r1	r←r∪r1	×	×	×	○
	mem	|= A	(mem)←(mem)∪A	×	×	×	○
	CY	|= [HL].bit	CY←CY∪(HL).bit	○	×	×	×
	CY	|= saddr.bit	CY←CY∪(saddr.bit)	○	○	○	○
	CY	|= /saddr.bit	CY←CY∪$\overline{(saddr.bit)}$	×	○	○	○
	CY	|= sfr.bit	CY←CY∪sfr.bit	○	○	○	○
	CY	|= /sfr.bit	CY←CY∪$\overline{sfr.bit}$	×	○	○	○
	CY	|= A.bit	CY←CY∪A.bit	○	○	○	○
	CY	|= /A.bit	CY←CY∪$\overline{A.bit}$	×	○	○	○
	CY	|= X.bit	CY←CY∪X.bit	×	○	○	○
	CY	|= /X.bit	CY←CY∪$\overline{X.bit}$	×	○	○	○
	CY	|= PSW.bit	CY←CY∪PSW.bit	○	○	○	×
	CY	|= /PSW.bit	CY←CY∪$\overline{PSW.bit}$	×	○	○	×
	CY	|= PSWH.bit	CY←CY∪PSW_{H}.bit	×	×	×	○
	CY	|= /PSWH.bit	CY←CY∪$\overline{PSW_{H}.bit}$	×	×	×	○
	CY	|= PSWL.bit	CY←CY∪PSW_{L}.bit	×	×	×	○
	CY	|= /PSWL.bit	CY←CY∪$\overline{PSW_{L}.bit}$	×	×	×	○

B2. Expression Statements <78K/0/I/II/III> (contd)

	Expression	Operation	0	I	II	III
XOR & Substitute	A ^= r	A←A ∀ r	○	×	×	×
	A ^= [HL]	A←A ∀ (HL)	○	×	×	×
	A ^= !addr16	A←A ∀ (!addr16)	○	○	×	×
	A ^= [HL + byte]	A←A ∀ (HL + byte)	○	×	×	×
	A ^= [HL + B]	A←A ∀ (HL + B)	○	×	×	×
	A ^= [HL + C]	A←A ∀ (HL + C)	○	×	×	×
	A ^= #byte	A←A ∀ byte	○	×	×	×
	A ^= saddr	A←A ∀ (saddr)	○	○	○	○
	A ^= sfr	A←A ∀ sfr	○	○	○	○
	A ^= mem	A←A ∀ (mem)	×	○	○	○
	A ^= &mem	A←A ∀ (&mem)	×	○	○	○
	saddr ^= #byte	(saddr)←(saddr) ∀ byte	×	×	○	×
	saddr ^= saddr	(saddr)←(saddr) ∀ (saddr)	○	○	○	○
	sfr ^= #byte	sfr←sfr ∀ byte	×	○	○	○
	r ^= A	r←r ∀ A	×	×	○	○
	r ^= r	r←r ∀ r	×	×	○	×
	r ^= r1	r←r ∀ r1	×	×	×	○
	mem ^= A	(mem)←(mem) ∀ A	×	×	×	○
	CY ^= [HL].bit	CY←CY ∀ (HL).bit	×	×	×	×
	CY ^= [r4]	CY←CY ∀ (r4)	○	○	×	×
	CY ^= saddr.bit	CY←CY ∀ (saddr.bit)	○	○	○	○
	CY ^= sfr.bit	CY←CY ∀ sfr.bit	○	○	○	○
	CY ^= A.bit	CY←CY ∀ A.bit	○	○	○	○
	CY ^= X.bit	CY←CY ∀ X.bit	×	○	○	○
	CY ^= PSW.bit	CY←CY ∀ PSW.bit	○	○	○	×
	CY ^= PSWH.bit	CY←CY ∀ PSW_{H}.bit	×	×	×	○
	CY ^= PSWL.bit	CY←CY ∀ PSW_{L}.bit	×	×	×	○

B2. Expression Statements <78K/0/I/II/III> (contd)

	Expression	Operation	0	I	II	III
Shift Right & Substitute	A >>= 1	(CY←A_0, A_7←0, A_{n-1}←A_n) × 1 time	○	×	×	×
	r >>= n	(CY←r_0, r_7←0, r_{n-1}←r_n) × n times	×	○	○	×
	rl >>= n	(CY←rl_0, rl_7←0, rl_{n-1}←rl_n) × n times	×	×	×	○
	rp >>= n	(CY←rp_0, rp_{15}←0, rp_{n-1}←rp_n) × n times	×	○	○	×
	rpl >>= n	(CY←rpl_0, rpl_{15}←0, rpl_{n-1}←rpl_n) × n times	×	×	×	○
Shift Left & Substitute	A <<= 1	(CY←A_7, A_0←0, A_{n+1}←A_n) × 1 time	○	×	×	×
	r <<= n	(CY←r_7, r_0←0, r_{n+1}←r_n) × n times	×	○	○	×
	rl <<= n	(CY←rl_7, rl_0←0, rl_{n+1}←rl_n) × n times	×	×	×	○
	rp <<= n	(CY←rp_{15}, rp_0←0, rp_{n+1}←rp_n) × n times	×	○	○	×
	rpl <<= n	(CY←rpl_{15}, rpl_0←0, rpl_{n+1}←rpl_n) × n times	×	×	×	○
Increment	r ++	r←r+1	○	○	○	×
	rl ++	rl←rl+1	×	×	×	○
	saddr ++	(saddr)←(saddr)+1	○	○	○	○
	rp ++	rp←rp+1	○	○	○	×
	rp2 ++	rp2←rp2+1	×	×	×	○
	saddrp ++	(saddrp)←(saddrp)+1	×	×	×	○
Decrement	r --	r←r-1	○	○	○	×
	rl --	rl←rl-1	×	×	×	○
	saddr --	(saddr)←(saddr)-1	○	○	○	○
	rp --	rp←rp-1	○	○	○	×
	rp2 --	rp2←rp2-1	×	×	×	○
	saddrp --	(saddrp)←(saddrp)-1	×	×	×	○
Exchange	A <-> [DE]	A←→(DE)	○	×	×	×
	A <-> [HL]	A←→(HL)	○	×	×	×
	A <-> !addr16	A←→(!addr16)	○	×	×	×
	A <-> [HL+byte]	A←→(HL+byte)	○	×	×	×
	A <-> [HL+B]	A←→(HL+B)	○	×	×	×
	A <-> [HL+C]	A←→(HL+C)	○	×	×	×
	A <-> [r4]	A←→(r4)	×	○	×	×
	A <-> r	A←→r	○	○	○	×
	A <-> rl	A←→rl	×	×	×	○
	A <-> mem	A←→(mem)	×	×	○	○
	A <-> &mem	A←→(&mem)	×	×	○	×
	A <-> saddr	A←→(saddr)	×	×	○	○
	A <-> sfr	A←→sfr	×	×	○	○
	A <-> [saddrp]	A←→((saddrp))	○	○	×	○
	r <-> r	r←→r	○	○	○	×
	r <-> rl	r←→rl	×	×	×	○
	saddr <-> saddr	(saddr)←→(saddr)	×	×	○	○

B-2. Expression Statements <78KVI>

Expression			Operation
Substitute/Substitute with Register name specification	br	= #byte	br←byte
	br	= br'	br←br'
	br	= mem	br←(mem)
	brl	= bsaddr	br←(bsaddr)
	brl	= [wsaddr]	br←((wsaddr))
	brl	= bsfr	br←bsfr
	brl	= meml	br←(meml)
	brl	= brl (bsfr)	brl←brl
	br2	= bsfr	br2←bsfr
	bsaddr	= #byte	(bsaddr)←byte
	bsaddr	= brl	(bsaddr)←brl
	bsaddr	= bsaddr'	(bsaddr)←(bsaddr')
	bsaddr	= br' (brl)	(bsaddr)←br'
	bsaddr	= bsaddr (brl)	(bsaddr)←(bsaddr')
	bsaddr	= [wsaddr] (brl)	(bsaddr)←((wsaddr))
	bsaddr	= bsfr (brl)	(bsaddr)←bsfr
	bsaddr	= mem (brl)	(bsaddr)←(mem)
	bsaddr	= meml (brl)	(bsaddr)←(meml)
	[wsaddr]	= brl	((wsaddr))←brl
	[wsaddr]	= #byte (brl)	((wsaddr))←byte
	[wsaddr]	= br' (brl)	((wsaddr))←br'
	[wsaddr]	= bsaddr (brl)	((wsaddr))←(bsaddr)
	[wsaddr]	= [wsaddr] (brl)	((wsaddr))←((wsaddr))
	[wsaddr]	= bsfr (brl)	((wsaddr))←bsfr
	[wsaddr]	= mem (brl)	((wsaddr))←(mem)
	[wsaddr]	= meml (brl)	((wsaddr))←(meml)
	bsfr	= #byte	bsf←byte
	bsfr	= brl	bsf←brl
	bsfr	= br2	bsf←br2
	bsfr	= br' (brl)	bsf←br'
	bsfr	= bsaddr (brl)	bsf←(bsaddr)
	bsfr	= [wsaddr] (brl)	bsf←((wsaddr))
	bsfr	= bsfr (brl)	bsf←bsfr
	bsfr	= mem (brl)	bsf←(mem)
	bsfr	= meml (brl)	bsf←(meml)
	mem	= #byte	(mem)←byte
	mem	= br	(mem)←br
	mem	= br' (br)	(mem)←br'
	mem	= mem (br)	(mem)←(mem)
	mem	= #word	(meml)←word
	mem	= wr	(meml)←wr
	mem	= wr' (wr)	(meml)←wr'
	mem	= wsaddr (wr)	(meml)←(wsaddr)
	mem	= [wsaddr] (wr)	(meml)←((wsaddr))

B-2. Expression Statements <78KVI>

	Expression	Operation
Substitute/Substitute with Register name specification	mem = wsfr (wr)	(mem)←wsfr
	mem = mem (wr)	(mem)←(mem)
	mem = mem1 (wr)	(mem)←(mem1)
	mem = PSW (wr)	(mem)←PSW
	mem = SP (wr)	(mem)←SP
	mem1 = #byte	(mem1)←byte
	mem1 = br1	(mem1)←br1
	mem1 = br' (br1)	(mem1)←br'
	mem1 = bsaddr (br1)	(mem1)←bsaddr
	mem1 = [wsaddr] (br1)	(mem1)←((wsaddr))
	mem1 = bsfr (br1)	(mem1)←bsfr
	mem1 = mem (br1)	(mem1)←(mem)
	mem1 = mem1 (br1)	(mem1)←(mem1)
	mem1 = #word	(mem1)←word
	mem1 = wr	(mem1)←wr
	mem1 = wr' (wr)	(mem1)←wr'
	mem1 = wsaddr (wr)	(mem1)←(wsaddr)
	mem1 = [wsaddr] (wr)	(mem1)←((wsaddr))
	mem1 = wsfr (wr)	(mem1)←wsfr
	mem1 = mem (wr)	(mem1)←(mem)
	mem1 = mem1 (wr)	(mem1)←(mem1)
	mem1 = PSW (wr)	(mem1)←PSW
	mem1 = SP (wr)	(mem1)←SP
	STBC = #byte	STBC←byte
	WDM = #byte	WDM←byte
	wr = #word	wr←word
	wr = wr'	wr←wr'
	wr = wsaddr	wr←(wsaddr)
	wr = [wsaddr]	wr←((wsaddr))
	wr = wsfr	wr←wsfr
	wr = mem	wr←(mem)
	wr = mem1	wr←(mem1)
	wr = PSW	wr←PSW
	wr = SP	wr←SP
	wr = wr3 (wsfr)	wr←wr3
	wr3 = wsfr	wr3←wsfr
	wsaddr = #word	(wsaddr)←word
	wsaddr = wr	(wsaddr)←wr
	wsaddr = wsaddr'	(wsaddr)←(wsaddr')
	wsaddr = wr' (wr)	(wsaddr)←wr'
	wsaddr = wsaddr (wr)	(wsaddr)←(wsaddr)
	wsaddr = [wsaddr] (wr)	(wsaddr)←((wsaddr))
	wsaddr = wsfr (wr)	(wsaddr)←wsfr
	wsaddr = mem (wr)	(wsaddr)←(mem)

B-2. Expression Statements <78KVI> (contd)

	Expression			Operation
Substitute/Substitute with Register name specification	wsaddr	=	meml (wr)	(wsaddr)←(meml)
	wsaddr	=	PSW (wr)	(wsaddr)←PSW
	wsaddr	=	SP (wr)	(wsaddr)←SP
	[wsaddr]	=	wr	((wsaddr))←wr
	[wsaddr]	=	#word (wr)	((wsaddr))←word
	[wsaddr]	=	wr' (wr)'	((wsaddr))←wr'
	[wsaddr]	=	wsaddr (wr)	((wsaddr))←(wsaddr)
	[wsaddr]	=	[wsaddr] (wr)	((wsaddr))←((wsaddr))
	[wsaddr]	=	wsfr (wr)	((wsaddr))←wsfr
	[wsaddr]	=	mem (wr)	((wsaddr))←(mem)
	[wsaddr]	=	meml (wr)	((wsaddr))←(meml)
	[wsaddr]	=	PSW (wr)	((wsaddr))←PSW
	[wsaddr]	=	SP (wr)	((wsaddr))←SP
	wsfr	=	#word	wsfr←word
	wsfr	=	wr	wsfr←wr
	wsfr	=	wr3	wsfr←wr3
	wsfr	=	wr' (wr)	wsfr←wr'
	wsfr	=	wsaddr (wr)	wsfr←(wsaddr)
	wsfr	=	[wsaddr] (wr)	wsfr←((wsaddr))
	wsfr	=	wsfr (wr)	wsfr←wsfr
	wsfr	=	mem (wr)	wsfr←(mem)
	wsfr	=	meml (wr)	wsfr←(meml)
	wsfr	=	PSW (wr)	wsfr←PSW
	wsfr	=	SP (wr)	wsfr←SP
	SP	=	#word	SP←word
	SP	=	wr	SP←wr
	SP	=	wr' (wr)	SP←wr'
	SP	=	wsaddr (wr)	SP←(wsaddr)
	SP	=	[wsaddr] (wr)	SP←((wsaddr))
	SP	=	wsfr (wr)	SP←wsfr
	SP	=	mem (wr)	SP←(mem)
	SP	=	meml (wr)	SP←(meml)
	SP	=	PSW (wr)	SP←PSW
	SP	=	SP (wr)	SP←SP
	dsaddr	=	wrp	(dsaddr)←wrp
	dsaddr	=	#dword (wrp)	(dsaddr)←dword
	dsaddr	=	dsaddr (wrp)	(dsaddr)←(dsaddr)
	wrp	=	#dword	wrp←dword
	wrp	=	dsaddr	wrp←(dsaddr)
	CY	=	br.bit3	CY←br.bit3
	CY	=	br.br'	CY←br.br'
	CY	=	bsaddr.bit3	CY←bsaddr.bit3
	CY	=	bsaddr.brl	CY←bsaddr.brl
	CY	=	bsfr.bit3	CY←bsfr.bit3

B-2. Expression Statements <78KVI> (contd)

	Expression			Operation
Substitute/Substitute with Register name specification	CY	=	bsfr. brl	CY←bsfr. brl
	CY	=	wr.bit4	CY←wr. bit4
	CY	=	wr.br	CY←wr.br
	br. bit3	=	CY	br. bit3←CY
	br. bit3	=	br. bit3 (CY)	br. bit3←br. bit3
	br. bit3	=	br. br' (CY)	br. bit3←br. br'
	br. bit3	=	bsaddr. bit3 (CY)	br. bit3←(bsaddr). bit3
	br. bit3	=	bsaddr. brl (CY)	br. bit3←(bsaddr). brl
	br. bit3	=	bsfr. bit3 (CY)	br. bit3←bsfr. bit3
	br. bit3	=	bsfr. brl (CY)	br. bit3←bsfr. brl
	br. br'	=	CY	br. br' ←CY
	br. br'	=	br. bit3 (CY)	br. br' ←br. bit3
	br. br'	=	br. br' (CY)	br. br' ←br. br'
	br. br'	=	bsaddr. bit3 (CY)	br. br' ←(bsaddr). bit3
	br. br'	=	baddr.brl (CY)	br. br' ←(bsaddr). brl
	br. br'	=	bsfr. bit3 (CY)	br. br' ←bsfr. bit3
	br. br'	=	bsfr. brl (CY)	br. br' ←bsfr. brl
	bsaddr. bit3	=	CY	(bsaddr). bit3←CY
	bsaddr. bit3	=	br. bit3 (CY)	(bsaddr). bit3←br. bit3
	bsaddr. bit3	=	br. br' (CY)	(bsaddr). bit3←br. br'
	bsaddr. bit3	=	bsaddr. bit3 (CY)	(bsaddr). bit3←(bsaddr). bit3
	bsaddr. bit3	=	baddr. brl (CY)	(bsaddr). bit3←(bsaddr). brl
	bsaddr. bit3	=	bsfr. bit3 (CY)	(bsaddr). bit3←bsfr. bit3
	bsaddr. bit3	=	bsfr. brl (CY)	(bsaddr). bit3←bsfr. brl
	bsaddr. brl	=	CY	(bsaddr). brl←CY
	bsaddr. brl	=	br. bit3 (CY)	(bsaddr). brl←br. bit3
	bsaddr. brl	=	br. br' (CY)	(bsaddr). brl←br. br'
	bsaddr. brl	=	bsaddr. bit3 (CY)	(bsaddr). brl←(bsaddr). bit3
	bsaddr. brl	=	baddr. brl (CY)	(bsaddr). brl←(bsaddr). brl
	bsaddr. brl	=	bsfr. bit3 (CY)	(bsaddr). brl←bsfr. bit3
	bsaddr. brl	=	bsfr. brl (CY)	(bsaddr). brl←bsfr. brl
	bsfr. bit3	=	CY	bsfr. bit3←CY
	bsfr. bit3	=	br. bit3 (CY)	bsfr. bit3←br. bit3
	bsfr. bit3	=	br. br' (CY)	bsfr. bit3←br. br'
	bsfr. bit3	=	bsaddr. bit3 (CY)	bsfr. bit3←(bsaddr). bit3
	bsfr. bit3	=	baddr. brl (CY)	bsfr. bit3←(bsaddr). brl
	bsfr. bit3	=	bsfr. bit3 (CY)	bsfr. bit3←bsfr. bit3
	bsfr. bit3	=	bsfr. brl (CY)	bsfr. bit3←bsfr. brl
	bsfr. brl	=	CY	bsfr. brl←CY
	bsfr. brl	=	br. bit3 (CY)	bsfr. brl←br. bit3
	bsfr. brl	=	br. br' (CY)	bsfr. brl←br. br'
	bsfr. brl	=	bsaddr. bit3 (CY)	bsfr. brl←bsaddr. bit3
	bsfr. brl	=	baddr. brl (CY)	bsfr. brl←baddr. brl
	bsfr. brl	=	bsfr. bit3 (CY)	bsfr. brl←bsfr. bit3

B-2. Expression Statements <78KVI> (contd)

	Expression	Operation
Substitute/Substitute with Register name specification	bsfr.br1 = bsfr.br1 (CY)	(bsaddr).br1←bsfr.br1
	wr.bit4 = CY	wr.bit4←CY
	wr.bit4 = wr.bit4 (CY)	wr.bit4←wr.bit4
	wr.bit4 = wr.br (CY)	wr.bit4←wr.br
	wr.br = CY	wr.br←CY
	wr.br = wr.bit4 (CY)	wr.br←wr.bit4
	wr.br = wr.br (CY)	wr.br←wr.br
Add & Substitute	br += #byte	br, CY←br+byte
	br += br'	br, CY←br+br'
	br += mem	br, CY←br+(mem)
	br1 += bsaddr	br1, CY←br1+(bsaddr)
	br1 += bsfr	br1, CY←br1+bsfr
	br1 += mem1	br1, CY←br1+(mem1)
	bsaddr += #byte	(bsaddr), CY←(bsaddr)+byte
	bsaddr += bsaddr'	(bsaddr), CY←(bsaddr)+(bsaddr')
	bsaddr += br1	(bsaddr), CY←(bsaddr)+br1
	bsfr += #byte	bsfr, CY←bsfr+byte
	mem += br	(mem), CY←(mem)+br
	mem += #byte	(mem), CY←(mem)+byte
	mem1 += br1	(mem1), CY←(mem1)+br1
	mem1 += #byte	(mem1), CY←(mem1)+byte
	wr += #word	wr, CY←wr+word
	wr += wr'	wr, CY←wr+wr'
	wr += wsaddr	wr, CY←wr+(wsaddr)
	wr += wsfr	wr, CY←wr+wsfr
	wr += mem	wr, CY←wr+(mem)
	wr += mem1	wr, CY←wr+(mem1)
	wsaddr += #word	(wsaddr), CY←(wsaddr)+word
	wsaddr += wr	(wsaddr), CY←(wsaddr)+wr
	wsaddr += wsaddr'	(wsaddr), CY←(wsaddr)+(wsaddr')
	wsfr += #word	wsfr, CY←wsfr+word
	mem += wr	(mem), CY←(mem)+wr
	mem += #word	(mem), CY←(mem)+word
	mem1 += wr	(mem1), CY←(mem1)+wr
	mem1 += #word	(mem1), CY←(mem1)+word
	SP += #word	SP←SP+word
	wrp += #dword	wrp, CY←wrp+dword
	wrp += wrp'	wrp, CY←wrp+wrp'
	wrp += dsaddr	wrp, CY←wrp+(dsaddr)
	dsaddr += wrp	(dsaddr), CY←(dsaddr)+wrp

Note 1

Note 1. If Add or Subtract with Carry is to be performed, describe the assembly language without change.

B-2. Expression Statements <78KVI> (contd)

	Expression			Operation
Note 1	Subtract & Substitute	br	−= #byte	br, CY←br−byte
		br	−= br'	br, CY←br−br'
		br	−= mem	br, CY←br−(mem)
		br1	−= bsaddr	br1, CY←br1−(bsaddr)
		br1	−= bsfr	br1, CY←br1−bsfr
		br1	−= mem1	br1, CY←br1−(mem1)
		bsaddr	−= #byte	(bsaddr), CY←(bsaddr)−byte
		bsaddr	−= bsaddr'	(bsaddr), CY←(bsaddr)−(bsaddr')
		bsaddr	−= br1	(bsaddr), CY←(bsaddr)−br1
		bsfr	−= #byte	bsfr, CY←bsfr−byte
		mem	−= br	(mem), CY←(mem)−br
		mem	−= #byte	(mem), CY←(mem)−byte
		mem1	−= br1	(mem1), CY←(mem1)−br1
		mem1	−= #byte	(mem1), CY←(mem1)−byte
		wr	−= #word	wr, CY←wr−word
		wr	−= wr'	wr, CY←wr−wr'
		wr	−= wsaddr	wr, CY←wr−(wsaddr)
		wr	−= wsfr	wr, CY←wr−wsfr
		wr	−= mem	wr, CY←wr−(mem)
		wr	−= mem1	wr, CY←wr−(mem1)
		wsaddr	−= #word	(wsaddr), CY←(wsaddr)−word
		wsaddr	−= wr	(wsaddr), CY←(wsaddr)−wr
		wsaddr	−= wsaddr'	(wsaddr), CY←(wsaddr)−(wsaddr')
		wsfr	−= #word	wsfr, CY←wsfr−word
		mem	−= wr	(mem), CY←(mem)−wr
		mem	−= #word	(mem), CY←(mem)−word
		mem1	−= wr	(mem1), CY←(mem1)−wr
		mem1	−= #word	(mem1), CY←(mem1)−word
		SP	−= #word	SP←SP−word
		wrp	−= #dword	wrp, CY←wrp−dword
		wrp	−= wrp'	wrp, CY←wrp−wrp'
		wrp	−= dsaddr	wrp, CY←wrp−(dsaddr)
		dsaddr	−= wrp	(dsaddr), CY←(dsaddr)−wrp
Note 2	Multiply & Substitute	wr	*= #byte	wr←wr_L×byte
		wr	*= br	wr←wr_L×br
		wr	*= bsaddr	wr←wr_L×(bsaddr)
		wr	*= mem	wr←wr_L×(mem)
		wrp	*= #word	wrp←wrp_L×word
		wrp	*= wr	wrp←wrp_L×wr
		wrp	*= wsaddr	wrp←wrp_L×(wsaddr)
		wrp	*= mem	wrp←wrp_L×(mem)

Note: 1. If Add or Subtract with Carry is to be performed, describe the assembly language without change.
2. With a signed Multiply or Divide instruction, describe the assembly language without change.

B-2. Expression Statements <78KVI> (contd)

	Expression			Operation
Note 2	Divide & Substitute	wr	/= #byte	wr(quo),ROL(rem) ← wr÷byte
		wr	/= br	wr(quo),br(rem) ← wr÷ br
		wr	/= bsaddr	wr(quo),bsaddr(rem) ← wr÷(bsaddr)
		wr	/= mem	wr(quo),mem(rem) ← wr÷(mem)
		wrp	/= #word	wrp(quo),ROL(rem) ← wrp÷word
		wrp	/= wr	wrp(quo),wr(rem) ← wrp÷ wr
		wrp	/= wsaddr	wrp(quo),wsaddr(rem) ← wrp÷(wsaddr)
		wrp	/= mem	wrp(quo),mem(rem) ← wrp÷(mem)
	AND & Substitute	br	&= #byte	br ← br ∩ byte
		br	&= br'	br ← br ∩ br'
		br	&= mem	br ← br ∩ (mem)
		br1	&= bsadd	br1 ← br1 ∩ (bsaddr)
		br1	&= bsfr	br1 ← br1 ∩ bsfr
		br1	&= mem1	br1 ← br1 ∩ (mem1)
		bsaddr	&= #byte	(bsaddr) ← (bsaddr) ∩ byte
		bsaddr	&= bsaddr'	(bsaddr) ← (bsaddr) ∩ (bsaddr')
		bsaddr	&= br1	(bsaddr) ← (bsaddr) ∩ br1
		bsfr	&= #byte	bsfr ← bsfr ∩ byte
		mem	&= br	(mem) ← (mem) ∩ br
		mem	&= #byte	(mem) ← (mem) ∩ byte
		mem1	&= br1	(mem1) ← (mem1) ∩ br1
		mem1	&= #byte	(mem1) ← (mem1) ∩ byte
		wr	&= #word	wr ← wr ∩ word
		wr	&= wr'	wr ← wr ∩ wr'
		wr	&= wsaddr	wr ← wr ∩ (wsaddr)
		wr	&= wsfr	wr ← wr ∩ wsfr
		wr	&= mem	wr ← wr ∩ (mem)
		wr	&= mem1	wr ← wr ∩ (mem1)
		wsaddr	&= #word	(wsaddr) ← (wsaddr) ∩ word
		wsaddr	&= wsaddr'	(wsaddr) ← (wsaddr) ∩ (wsaddr')
		wsaddr	&= wr	(wsaddr) ← (wsaddr) ∩ wr
		wsfr	&= #word	wsfr ← wsfr ∩ word
		mem	&= wr	(mem) ← (mem) ∩ wr
		mem	&= #word	(mem) ← (mem) ∩ word
		mem1	&= wr	(mem1) ← (mem1) ∩ wr
		mem1	&= #word	(mem1) ← (mem1) ∩ word
		wrp	&= #dword	wrp ← wrp ∩ dword
		wrp	&= wrp'	wrp ← wrp ∩ wrp'
		wrp	&= dsaddr	wrp ← wrp ∩ (dsaddr)
		dsaddr	&= wrp	(dsaddr) ← (dsaddr) ∩ wrp
		CY	&= br.bit3	CY ← CY ∩ br.bit3
		CY	&= /br.bit3	CY ← CY ∩ $\overline{\text{br.bit3}}$
		CY	&= br.br'	CY ← CY ∩ br.br'

quo: Quotient; rem: Remainder

Note 2. With a signed Multiply instruction, describe the assembly language without change.

B-2. Expression Statements <78KVI> (contd)

	Expression	Operation
AND & Substitute	CY & = /br.br'	CY←CY ∩ $\overline{br.br'}$
	CY & = bsaddr.bit3	CY←CY ∩ (bsaddr).bit3
	CY & = /bsaddr.bit3	CY←CY ∩ $\overline{(bsaddr).bit3}$
	CY & = bsaddr.br1	CY←CY ∩ (saddr).br1
	CY & = /bsaddr.br1	CY←CY ∩ $\overline{(bsaddr).br1}$
	CY & = bsfr.bit3	CY←CY ∩ bsfr.bit3
	CY & = /bsfr.bit3	CY←CY ∩ $\overline{bsfr.bit3}$
	CY & = bsfr.br1	CY←CY ∩ bsfr.br1
	CY & = /bsfr.br1	CY←CY ∩ $\overline{bsfr.br1}$
	CY & = wr.bit4	CY←CY ∩ wr.bit4
	CY & = /wr.bit4	CY←CY ∩ $\overline{wr.bit4}$
	CY & = wr.br	CY←CY ∩ wr.br
	CY & = /wr.br	CY←CY ∩ $\overline{wr.br}$
OR & Substitute	br | = #byte	br←br ∪byte
	br | = br'	br←br ∪ br'
	br | = mem	br←br ∪ (mem)
	br1 | = bsaddr	br1←br1 ∪ (bsaddr)
	br1 | = bsfr	br1←br1 ∪ bsfr
	br1 | = mem1	br1←br1 ∪ (mem1)
	bsaddr | = #byte	(bsaddr)←(bsaddr) ∪byte
	bsaddr | = bsaddr'	(bsaddr)←(bsaddr) ∪ (bsaddr')
	bsaddr | = br1	(bsaddr)←(bsaddr) ∪ br1
	bsfr | = #byte	bsfr←bsfr ∪byte
	mem | = br	(mem)←(mem) ∪ br
	mem | = #byte	(mem)←(mem) ∪byte
	mem1 | = br1	(mem1)←(mem1) ∪ br1
	mem1 | = #byte	(mem1)←(mem1) ∪byte
	wr | = #word	wr←wr ∪word
	wr | = wr'	wr←wr ∪ wr'
	wr | = wsaddr	wr←wr ∪ (wsaddr)
	wr | = wsfr	wr←wr ∪ wsfr
	wr | = mem	wr←wr ∪ (mem)
	wr | = mem1	wr←wr ∪ (mem1)
	wsaddr | = #word	(wsaddr)←(wsaddr) ∪word
	wsaddr | = wsaddr'	(wsaddr)←(wsaddr) ∪ (wsaddr')
	wsaddr | = wr	(wsaddr)←(wsaddr) ∪ wr
	wsfr | = #word	wsfr←wsfr ∪word
	mem | = wr	(mem)←(mem) ∪ wr
	mem | = #word	(mem)←(mem) ∪word
	mem1 | = wr	(mem1)←(mem1) ∪ wr
	mem1 | = #word	(mem1)←(mem1) ∪word
	wrp | = # dword	wrp←wrp ∪ dword
	wrp | = wrp'	wrp←wrp ∪ wrp'
	wrp | = dsaddr	wrp←wrp ∪ (dsaddr)

B-2. Expression Statements <78KVI> (contd)

	Expression			Operation
OR & Substitute	dsaddr	| =	wrp	(dsaddr)←(dsaddr) ∪ wrp
	CY	| =	br.bit3	CY←CY ∪ br.bit3
	CY	| =	/br.bit3	CY←CY ∪ $\overline{\text{br.bit3}}$
	CY	| =	br.br'	CY←CY ∪ br.br'
	CY	| =	/br.br'	CY←CY ∪ $\overline{\text{br.br'}}$
	CY	| =	bsaddr.bit3	CY←CY ∪ (bsaddr).bit3
	CY	| =	/bsaddr.bit3	CY←CY ∪ $\overline{\text{(bsaddr).bit3}}$
	CY	| =	bsaddr.br1	CY←CY ∪ (saddr).br1
	CY	| =	/bsaddr.br1	CY←CY $\overline{\text{(bsaddr).br1}}$
	CY	| =	bsfr.bit3	CY←CY ∪ bsfr.bit3
	CY	| =	/bsfr.bit3	CY←CY ∪ $\overline{\text{bsfr.bit3}}$
	CY	| =	bsfr.br1	CY←CY ∪ bsfr.br1
	CY	| =	/bsfr.br1	CY←CY ∪ $\overline{\text{bsfr.br1}}$
	CY	| =	wr.bit4	CY←CY ∪ wr.bit4
	CY	| =	/wr.bit4	CY←CY ∪ $\overline{\text{wr.bit4}}$
	CY	| =	wr.br	CY←CY ∪ wr.br
	CY	| =	/wr.br	CY←CY ∪ $\overline{\text{wr.br}}$
XOR & Substitute	br	^ =	#byte	br←br ∀byte
	br	^ =	br'	br←br ∀ br'
	br	^ =	mem	br←br ∀(mem)
	br1	^ =	bsaddr	br1←br1 ∀(bsaddr)
	br1	^ =	bsfr	br1←br1 ∀ bsfr
	br1	^ =	mem1	br1←br1 ∀(mem1)
	bsaddr	^ =	#byte	(bsaddr)←(bsaddr)∀byte
	bsaddr	^ =	bsaddr'	(bsaddr)←(bsaddr)∀(bsaddr')
	bsaddr	^ =	br1	(bsaddr)←(bsaddr)∀ br1
	bsfr	^ =	#byte	bsfr←bsfr ∀byte
	mem	^ =	br	(mem)←(mem)∀ br
	mem	^ =	#byte	(mem)←(mem)∀byte
	mem1	^ =	br1	(mem1)←(mem1)∀ br1
	mem1	^ =	#byte	(mem1)←(mem1)∀byte
	wr	^ =	#word	wr←wr ∀word
	wr	^ =	wr'	wr←wr ∀ wr'
	wr	^ =	wsaddr	wr←wr ∀(wsaddr)
	wr	^ =	wsfr	wr←wr ∀ wsfr
	wr	^ =	mem	wr←wr ∀(mem)
	wr	^ =	mem1	wr←wr ∀(mem1)
	wsaddr	^ =	#word	(wsaddr)←(wsaddr)∀word
	wsaddr	^ =	wsaddr'	(wsaddr)←(wsaddr)∀(wsaddr')
	wsaddr	^ =	wr	(wsaddr)←(wsaddr)∀ wr
	wsfr	^ =	#word	wsfr←wsfr ∀word
	mem	^ =	wr	(mem)←(mem)∀ wr
	mem	^ =	#word	(mem)←(mem)∀word

B-2. Expression Statements <78KVI> (contd)

	Expression			Operation
XOR & Substitute	mem1	^=	wr	(mem1)←(mem1)∀wr
	mem1	^=	#word	(mem1)←(mem1)∀word
	wrp	^=	#dword	wrp←wrp∀dword
	wrp	^=	wrp'	wrp←wrp∀wrp'
	wrp	^=	dsaddr	wrp←wrp∀(dsaddr)
	dsaddr	^=	wrp	(dsaddr)←(dsaddr)∀wrp
	CY	^=	br.bit3	CY←CY∀br.bit3
	CY	^=	br.br'	CY←CY∀br.br'
	CY	^=	bsaddr.bit3	CY←CY∀(bsaddr).bit3
	CY	^=	bsaddr.br1	CY←CY∀(saddr).br1
	CY	^=	bsfr.bit3	CY←CY∀bsfr.bit3
	CY	^=	bsfr.br1	CY←CY∀bsfr.br1
	CY	^=	wr.bit4	CY←CY∀wr.bit4
	CY	^=	wr.br	CY←CY∀wr.br

B-2. Expression Statements <78KVI> (contd)

	Expression	Operation
Shift Right & Substitute	br >>= n3	{CY←br_0, br_7←0, br_{n-1}←br_n} × n3 times
	br >>= br1	{CY←br_0, br_7←0, br_{n-1}←br_n} × br1 times
	bsaddr >>= 1	CY←(bsaddr)$_0$, (bsaddr)$_7$←0, (bsaddr)$_{n-1}$←(bsaddr)$_n$
	bsaddr >>= br1	{CY←(bsaddr)$_0$, (bsaddr)$_7$←0, (bsaddr)$_{n-1}$←(bsaddr)$_n$} × br1 times
	mem >>= 1	CY←(mem)$_0$, (mem)$_7$←0, (mem)$_{n-1}$←(mem)$_n$, and CY←(mem)$_0$, (mem)$_{15}$←0, (mem)$_{n-1}$←(mem)$_n$
	mem >>= br1	{CY←(mem)$_0$, (mem)$_7$←0, (mem)$_{n-1}$←(mem)$_n$} × br1 times and {CY←(mem)$_0$, (mem)$_{15}$←0, (mem)$_{n-1}$←(mem)$_n$} × br1 times
	wr >>= n4	{CY←wr_0, wr_{15}←0, wr_{n-1}←wr_n} × n4 times
	wr >>= br1	{CY←wr_0, wr_{15}←0, wr_{n-1}←wr_n} × br1 times
	wsaddr >>= 1	CY←(wsaddr)$_0$, (wsaddr)$_{15}$←0, (wsaddr)$_{n-1}$←(wsaddr)$_n$
	wsaddr >>= br1	{CY←(wsaddr)$_0$, (wsaddr)$_{15}$←0, (wsaddr)$_{n-1}$←(wsaddr)$_n$} × br1 times
	wrp >>= n5	{CY←wrp_0, wrp_{31}←0, wrp_{n-1}←wrp_n} × br1 times
	wrp >>= br1	{CY←wrp_0, wrp_{31}←0, wrp_{n-1}←wrp_n} × n5 times
Shift Left & Substitute	br <<= n3	{CY←br_7, br_0←0, br_n←br_{n-1}} × n3 times
	br <<= br1	{CY←br_7, br_0←0, br_n←br_{n-1}} × br1 times
	bsaddr <<= 1	CY←(bsaddr)$_7$, (bsaddr)$_0$←0, (bsaddr)$_n$←(bsaddr)$_{n-1}$
	bsaddr <<= br1	{CY←(bsaddr)$_7$, (bsaddr)$_0$←0, (bsaddr)$_n$←(bsaddr)$_{n-1}$} × br1 times
	mem <<= 1	CY←(mem)$_7$, (mem)$_0$←0, (mem)$_n$←(mem)$_{n-1}$ and CY←(mem)$_{15}$, (mem)$_0$←0, (mem)$_n$←(mem)$_{n-1}$
	mem <<= br1	{CY←(mem)$_7$, (mem)$_0$←0, (mem)$_n$←(mem)$_{n-1}$} × br1 times and {CY←(mem)$_{15}$, (mem)$_0$←0, (mem)$_n$←(mem)$_{n-1}$} × br1 times
	wr <<= n4	{CY←wr_{15}, wr_0←0, wr_n←wr_{n-1}} × n4 times
	wr <<= br1	{CY←wr_{15}, wr_0←0, wr_n←wr_{n-1}} × br1 times
	wsaddr <<= 1	CY←(wsaddr)$_{15}$, (wsaddr)$_0$←0, (wsaddr)$_n$←(wsaddr)$_{n-1}$
	wsaddr <<= br1	{CY←(wsaddr)$_{15}$, (wsaddr)$_0$←0, (wsaddr)$_n$←(wsaddr)$_{n-1}$} × br1 times
	wrp <<= n5	{CY←wrp_{31}, wrp_0←0, wrp_n←wrp_{n-1}} × n5 times
	wrp <<= br1	{CY←wrp_{31}, wrp_0←0, wrp_n←wrp_{n-1}} × br1 times

B-2. Expression Statements <78KVI> (contd)

	Expression	Operation
Increment	br++	br←br+1
	bsaddr++	(bsaddr)←(bsaddr)+1
	mem++	(mem)←(mem)+1
	mem1++	(mem1)←(mem1)+1
	wr++	wr←wr+1
	wsaddr++	(wsaddr)←(wsaddr)+1
	SP++	SP←SP+1
Decrement	br--	br←br-1
	bsaddr--	(bsaddr)←(bsaddr)-1
	mem--	(mem)←(mem)-1
	mem1--	(mem1)←(mem1)-1
	wr--	wr←wr-1
	wsaddr--	(wsaddr)←(wsaddr)-1
	SP--	SP←SP-1
Exchange	br <-> br'	br⟷br'
	br <-> mem	br⟷(mem)
	bsaddr <-> bsaddr'	(bsaddr)⟷(bsaddr')
	brl <-> bsaddr'	brl⟷(bsaddr')
	brl <-> [wsaddr]	brl⟷((wsaddr))
	brl <-> bsfr	brl⟷bsfr
	brl <-> mem1	brl⟷(mem1)
	wr <-> wr'	wr⟷wr'
	wr <-> wsaddr	wr⟷(wsaddr)
	wr <-> [wsaddr]	wr⟷((wsaddr))
	wr <-> wsfr	wr⟷wsfr
	wr <-> mem	wr⟷(mem)
	wr <-> mem1	wr⟷(mem1)
	wsaddr <-> wsaddr'	(wsaddr)⟷(wsaddr')

APPENDIX C. LIST OF GENERATED INSTRUCTIONS

C-1. Condition Expressions <78K0/I/II/III>

Relational expression	Instruction generated	Condition of control statement
α == β	CMP (W) α,β BNZ $??FALSE	Lowercase
	CMP (W) α,β BZ $??TRUE BR ??FALSE ??TRUE:	Uppercase
α == β (γ)	MOV (W) γ,α CMP (W) γ,β BNZ $??FALSE	Lowercase
	MOV (W) γ,α CMP (W) γ,β BZ $??TRUE BR ??FALSE ??TRUE:	Uppercase
α != β	CMP (W) α,β BZ $??FALSE	Lowercase
	CMP (W) α,β BNZ $??TRUE BR ??FALSE ??TRUE:	Uppercase
α != β (γ)	MOV (W) γ,α CMP (W) γ,β BZ $??FALSE	Lowercase
	MOV (W) γ,α CMP (W) γ,β BNZ $??TRUE BR ??FALSE ??TRUE:	Uppercase
α < β	CMP (W) α,β BNC $??FALSE	Lowercase
	CMP (W) α,β BC $??TRUE BR ??FALSE ??TRUE:	Uppercase
α < β (γ)	MOV (W) γ,α CMP (W) γ,β BNC $??FALSE	Lowercase
	MOV (W) γ,α CMP (W) γ,β BC $??TRUE BR ??FALSE ??TRUE:	Uppercase

C-1. Condition Expressions <78K0/I/II/III> (contd)

Relational expression		Instructions generated	Condition of control statement
α > β	0/I/II	CMP (W) α,β BZ $S??FALSE BC $S??FALSE	Lowercase
	0/I/II	CMP (W) α,β BZ S+4 BNC $S??TRUE BR ??FALSE ??TRUE:	Uppercase
	III	CMP (W) α,β BNH $S??FALSE	Lowercase
	III	CMP (W) α,β BH $S??TRUE BR ??FALSE ??TRUE:	Uppercase
α > β (γ)	0/I/II	MOV (W) γ,α CMP (W) γ,β BZ $S??FALSE BC $S??FALSE	Lowercase
	0/I/II	MOV (W) γ,α CMP (W) γ,β BZ S+4 BNC $S??TRUE BR ??FALSE ??TRUE:	Uppercase
	III	MOV (W) γ,α CMP (W) γ,β BNH $S??FALSE	Lowercase
	III	MOV (W) γ,α CMP (W) γ,β BH $S??TRUE BR ??FALSE ??TRUE:	Uppercase
α >= β		CMP (W) α,β BC $S??FALSE	Lowercase
		CMP (W) α,β BNC $S??TRUE BR ??FALSE ??TRUE:	Uppercase
α >= β (γ)		MOV (W) γ,α CMP (W) γ,β BC $S??FALSE	Lowercase

C-1. Condition Expressions <78K0/I/II/III> (contd)

Relational expression		Instructions generated	Condition of control statement
α >= β (γ)		MOV (W) γ,α CMP (W) γ,β BNC $??TRUE BR ??FALSE ??TRUE:	Uppercase
α <= β	0/I/II	CMP (W) α,β BZ $$+4 BNC $??FALSE	Lowercase
	0/I/II	CMP (W) α,β BZ $??TRUE BC $??TRUE BR ??FALSE ??TRUE:	Uppercase
	III	CMP (W) α,β BH $??FALSE	Lowercase
	III	CMP (W) α,β BNH $??TRUE BR ??FALSE ??TRUE:	Uppercase
α <= β (γ)	0/I/II	MOV (W) γ,α CMP (W) γ,β BZ $$+4 BNC $??FALSE	Lowercase
	0/I/II	MOV (W) γ,α CMP (W) γ,β BZ $??TRUE BC $??TRUE BR ??FALSE ??TRUE:	Uppercase
	III	MOV (W) γ,α CMP (W) γ,β BH $??FALSE	Lowercase
	III	MOV (W) γ,α CMP (W) γ,β BNH $??TRUE BR ??FALSE ??TRUE:	Uppercase

C-1. Condition Expressions <78K0/I/II/III> (contd)

Bit condition expression	Instructions generated		Condition of control statement
if_bit (bs) elseif_bit (bs) while_bit (bs) until_bit (bs) bs: Bit symbol	①	BNC $??FALSE	Lowercase (Bit symbol is CY.)
	②	BNZ $??FALSE	Lowercase (Bit symbol is Z.)
	③	BF bs $??FALSE	Lowercase (Bit symbol is other than ① & ② .)
	④	BC $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is CY)
	⑤	BZ $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is Z.)
	⑥	BT bs $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is other than ④ & ⑤ .)
if_bit (!bs) elseif_bit (!bs) while_bit (!bs) until_bit (!bs) bs: Bit symbol	①	BC $??FALSE	Lowercase (Bit symbol is CY.)
	②	BZ $??FALSE	Lowercase (Bit symbol is Z.)
	③	BT bs $??FALSE	Lowercase (Bit symbol is other than ① & ② .)
	④	BNC $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is CY)
	⑤	BNZ $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is Z.)
	⑥	BF bs $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is other than ④ & ⑤ .)

C-1. Condition Expressions <78K0/I/II/III>

<table>
<tr><th>Logical operator expression</th><th>Instructions generated</th><th>Condition of control statement</th></tr>
<tr><td rowspan="2">expression 1 &&
expression 2

Example:

α1 ==β1 && α2 !=β2</td><td>CMP (W) α1,β1
BNZ $??FALSE
CMP (W) α2,β2
BZ $??FALSE</td><td>Lowercase</td></tr>
<tr><td>CMP (W) α1,β1
BZ $??TRUE1
BR ??FALSE
??TRUE1:
CMP (W) α2,β2
BNZ $??TRUE2
BR ??FALSE
??TRUE2:</td><td>Uppercase</td></tr>
<tr><td rowspan="2">expression 1 ||
expression 2

Example:

α1 ==β1 || α2 !=β2</td><td>CMP (W) α1,β1
BZ $??TRUE
CMP (W) α2,β2
BZ $??FALSE</td><td>Lowercase</td></tr>
<tr><td>CMP (W) α1,β1
BZ $??TRUE
CMP (W) α2,β2
BNZ $??TRUE
BR ??FALSE
??TRUE:</td><td>Uppercase</td></tr>
</table>

C-1. Condition Expressions <78K0/VI>

Relational operator expression	Instructions generated	Condition of control statement
α == β	CMPB (W.D) α,β BNZS $??FALSE	Lowercase
	CMPB (W.D) α,β BZS $??TRUE BR ??FALSE ??TRUE:	Uppercase
α == β (γ)	MOVB (W.D) γ,α CMPB (W.D) γ,β BNZS $??FALSE	Lowercase
	MOVB (W.D) γ,α CMPB (W.D) γ,β BZS $??TRUE BR ??FALSE ??TRUE:	Uppercase
α != β	CMPB (W.D) α,β BZS $??FALSE	Lowercase
	CMPB (W.D) α,β BNZS $??TRUE BR ??FALSE ??TRUE:	Uppercase
α != β (γ)	MOVB (W.D) γ,α CMPB (W.D) γ,β BZS $??FALSE	Lowercase
	MOVB (W.D) γ,α CMPB (W.D) γ,β BNZS $??TRUE BR ??FALSE ??TRUE:	Uppercase
α < β	CMPB (W.D) α,β BNCS $??FALSE	Lowercase
	CMPB (W.D) α,β BCS $??TRUE BR ??FALSE ??TRUE:	Uppercase
α < β (γ)	MOVB (W.D) γ,α CMPB (W.D) γ,β BNCS $??FALSE	Lowercase
	MOVB (W.D) γ,α CMPB (W.D) γ,β BCS $??TRUE BR ??FALSE ??TRUE:	Uppercase

C-1. Condition Expressions <78KVI> (contd)

Relational operator expression	Instructions generated		Condition of control statement
$\alpha > \beta$	CMP (W.D)	α, β	Lowercase
	BNHS	$??FALSE	
	CMPB (W.D)	α, β	Uppercase
	BHS	$??TRUE	
	BR	??FALSE	
	??TRUE:		
$\alpha > \beta \quad (\gamma)$	MOVB (W.D)	γ, α	Lowercase
	CMPB (W.D)	γ, β	
	BNHS	$??FALSE	
	MOVB (W.D)	γ, α	Uppercase
	CMPB (W.D)	γ, β	
	BHS	$??TRUE	
	BR	??FALSE	
	??TRUE:		
$\alpha >= \beta$	CMPB (W.D)	α, β	Lowercase
	BCS	$??FALSE	
	CMPB (W.D)	α, β	Uppercase
	BNCS	$??TRUE	
	BR	??FALSE	
	??TRUE:		
$\alpha >= \beta \quad (\gamma)$	MOVB (W.D)	γ, α	Lowercase
	CMPB (W.D)	γ, β	
	BCS	$??FALSE	
$\alpha >= \beta \quad (\gamma)$	MOVB (W.D)	γ, α	Uppercase
	CMPB (W.D)	γ, β	
	BNCS	$??TRUE	
	BR	??FALSE	
	??TRUE:		
$\alpha <= \beta$	CMPB (W.D)	α, β	Lowercase
	BHS	$??FALSE	
	CMPB (W.D)	α, β	Uppercase
	BNHS	$??TRUE	
	BR	??FALSE	
	??TRUE:		
$\alpha <= \beta \quad (\gamma)$	MOVB (W.D)	γ, α	Lowercase
	CMPB (W.D)	γ, β	
	BHS	$??FALSE	
$\alpha <= \beta \quad (\gamma)$	MOVB (W.D)	γ, α	Uppercase
	CMPB (W.D)	γ, β	
	BNHS	$??TRUE	
	BR	??FALSE	
	??TRUE:		

C-1. Condition Expressions <78KVI> (contd)

Bit condition expression	Instructions generated		Condition of control statement
if_bit (bs) else_bit (bs) while_bit (bs) until_bit (bs) bs: Bit symbol	①	BNCS $??FALSE	Lowercase (Bit symbol is CY.)
	②	BNZS $??FALSE	Lowercase (Bit symbol is Z.)
	③	BFBS bs $??FALSE	Lowercase (Bit symbol is 8 bits and other than ① & ② .)
	④	BFWS bs $??FALSE	Lowercase (Bit symbol is 16 bits and other than ① & ② .)
	⑤	BCS $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is CY)
	⑥	BZS $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is Z.)
	⑦	BTBS bs $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is 8 bits and other than ④ & ⑤ .)
	⑧	BTWS bs $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is 16 bits and other than ④ & ⑤ .)
if_bit (!bs) else_bit (!bs) while_bit (!bs) until_bit (!bs) bs: Bit symbol	①	BCS $??FALSE	Lowercase (Bit symbol is CY.)
	②	BZS $??FALSE	Lowercase (Bit symbol is Z.)
	③	BTBS bs $??FALSE	Lowercase (Bit symbol is 8 bits and other than ① & ② .)
	④	BTWS bs $??FALSE	Lowercase (Bit symbol is 16 bits and other than ① & ② .)
	⑤	BNCS $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is CY)
	⑥	BNZS $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is Z.)
	⑦	BFBS bs $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is 8 bits and other than ④ & ⑤ .)
	⑧	BFWS bs $??TRUE BR ??FALSE ??TRUE:	Uppercase (Bit symbol is 16 bits and other than ④ & ⑤ .)

C-1. Condition Expressions <78KVI>

Logical operator expression	Instruction generated	条 件
expression 1 && expression 2 Example: α1 == β1 && α2 != β2	CMPB (W.D) α1, β1 BNZS $??FALSE CMPB (W.D) α2, β2 BZS $??FALSE	Lowercase
	CMPB (W.D) α1, β1 BZS $??TRUE1 BR ??FALSE ??TRUE1: CMPB (W.D) α2, β2 BNZS $??TRUE2 BR ??FALSE ??TRUE2:	Uppercase
expression 1 || expression 2 Example: α1 == β1 || α2 != β2	CMPB (W.D) α1, β1 BZS S??TRUE CMPB (W.D) α2, β2 BZS S??FALSE	Lowercase
	CMPB (W.D) α1, β1 BZS S??TRUE CMPB (W.D) α2, β2 BNZS S??TRUE BR ??FALSE ??TRUE:	Uppercase

C-2. Expression Statements (78K0/I/II/III>

Substitution statement	Instruction generated			Condition
α = β	①	MOV1 α,β		alpha or betea is CY.
	②	MOVW α,β		alpha or beta is a word in other than ① above.
	③	MOV α,β		In other than ① and ②.
α = β (γ)	①	MOV1 CY,β MOV1 α,CY		gamma is CY.
	②	MOVW γ,β MOVW α,γ		gamma is a word symbol in other than ① above.
	③	MOV γ,β MOV α,γ		In other than ① and ② .
α += β	①	ADDW α,β		alpha is a word symbol.
	②	ADD α,β		In other than ① above.
α -= β	①	SUBW α,β		alpha is a word symbol.
	②	SUB α,β		In other than ① above.
α *= β	0 II	①	MULU β	-
	I	①	MULUW β	-
	III	①	MULUW β	Beta is a register pair.
		②	MULU β	In other than ① above.
α /= β	0 I II	①	DIVUW β	-
	III	①	DIVUX β	Beta is a register pair.
		②	DIVUW β	In other than ① above.
α &= β	①	AND1 CY,β		alpha is CY.
	②	AND α,β		In other than ① above.
α |= β	①	OR1 CY,β		alpha is CY.
	②	OR α,β		In other than ① above.
α ^= β	①	XOR1 CY,β		alpha is CY.
	②	XOR α,β		In other than ① above.

C-2. Expression Statements (78K0/I/II/III> (contd)

Substitution statement	Instruction generated			Condition
α >>= β	0	①	ROR α,β	-
	I II III	①	SHRW α,β	alpha is a word symbol.
		②	SHR α,β	In other than ① above.
α <<= β	0	①	ROL α,β	-
	I II III	①	SHLW α,β	alpha is a word symbol.
		②	SHL α,β	In other than ① above.
α++	①		INCW α	alpha is a word symbol.
	②		INC α	In other than ① above.
α--	①		DECW α	alpha is a word symbol.
	②		DEC α	In other than ① above.
α <-> β	0 I II	①	XCH α,β	-
	III	①	XCHW α,β	alpha is a word symbol.
		②	XCH α,β	In other than ① above.

C-2. Expression Statements <78KVI>

Substitution statement	Instruction generated		Condition
α = β	①	MOV1W α,β	alpha or beta is CY and others are word symbols.
	②	MOV1B α,β	alpha or beta is CY in other than ①.
	③	MOVD α,β	alpha or beta is a double word symbol in other than ① and ②.
	④	MOVW α,β	alpha or beta is a word symbol in other than ①, ②, and ③ above.
	⑤	MOVB α,β	In other than ① thru ④.
α = β (γ)	①	MOV1W CY,β MOV1W α,CY	gamma is CY and alpha or beta is a word symbol.
	②	MOV1B CY,β MOV1B α,CY	gamma is CY in other than ① above.
	③	MOVD γ,β MOVD α,γ	gamma is a double word in other than ① and ②.
	④	MOVW γ,β MOVW α,γ	gamma is a word in other than ① thru ③.
	⑤	MOVB γ,β MOVB α,γ	In other than ① thru ④.
α += β	①	ADDD α,β	alpha or beta is a word register pair.
	②	ADDW α,β	alpha or beta is a word symbol.
	③	ADDB α,β	In other than ① and ②.
α -= β	①	SUBD α,β	alpha or beta is a word register pair.
	②	SUBW α,β	alpha or beta is a word symbol.
	③	SUBB α,β	In other than ① and ②.
α *= β	①	MULUW α,β	alpha is a word register pair.
	②	MULUB α,β	alpha is word register.
α /= β	①	DIVUD α,β	alpha is a word register pair.
	②	DIVUW α,β	alpha is word register.

C-2. Expression Statements <78KVI> (contd)

Substitution statement	Instruction generated		Condition
α &= β	①	AND1W CY,β	alpha is CY and beta is a word register.
	②	AND1B CY,β	alpha is CY and beta is a byte symbol.
	③	ANDD α,β	alpha or beta is a word register pair.
	④	ANDW α,β	alpha or beta is a word symbol in other than ① thru ③.
	⑤	ANDB α,β	In other than ① thru ④.
α |= β	①	OR1W CY,β	alpha is CY and beta is a word register.
	②	OR1B CY,β	alpha is CY and beta is a byte symbol.
	③	ORD α,β	alpha or beta is a word register pair.
	④	ORW α,β	alpha or beta is a word symbol in other than ① thru ③.
	⑤	ORB α,β	In other than ① thru ④.
α ^= β	①	XOR1W CY,β	alpha is CY and beta is a word register.
	②	XOR1B CY,β	alpha is CY and beta is a byte symbol.
	③	XORD α,β	alpha or beta is a word register pair.
	④	XORW α,β	alpha or beta is a word symbol in other than ① thru ③.
	⑤	XORB α,β	In other than ① thru ④.
α >>= β	①	SHRD α,β	alpha is a word register pair.
	②	SHRW α,β	alpha is a word symbol.
	③	SHRB α,β	In other than ① and ②.
α <<= β	①	SHLD α,β	alpha is a word register pair.
	②	SHLW α,β	alpha is a word symbol.
	③	SHLB α,β	In other than ① and ②.

C-2. Expression Statements <78KVI> (contd)

Substitution statement	Instruction generated		Condition
α++	①	INCW α	alpha is a word symbol.
	②	INCB α	In other than ① above.
α--	①	DECW α	alpha is a word symbol.
	②	DECB α	In other than ① above.
α <-> β	①	XCHW α,β	alpha is a word symbol.
	②	XCHB α,β	In other than ① above.

NEC