NEC

User's Manual

RA75X ASSEMBLER PACKAGE

Structured Assembler Preprocessor

Version 3.00

Target Devices:
75X Series
75XL Series

Document No. U12593EJ5V0UM00 (5th edition)
Previous No. EEU-1304
Date Published August 1997 N

[MEMO]

M7A 96.10

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 800-366-9782
Fax: 800-729-9288

NEC Electronics (Germany) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 02
Fax: 0211-65 03 490

NEC Electronics (UK) Ltd.
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Italiana s.r.1.
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

NEC Electronics (Germany) GmbH
Benelux Office
Eindhoven, The Netherlands
Tel: 040-2445845
Fax: 040-2444580

NEC Electronics (France) S.A.
Velizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

NEC Electronics (France) S.A.
Spain Office
Madrid, Spain
Tel: 01-504-2787
Fax: 01-504-2860

NEC Electronics (Germany) GmbH
Scandinavia Office
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
United Square, Singapore 1130
Tel: 253-8311
Fax: 250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-719-2377
Fax: 02-719-5951

NEC do Brasil S.A.
Sao Paulo-SP, Brasil
Tel: 011-889-1680
Fax: 011-889-1689

Major Revision in This Edition

Page	Description
Throughout	Addition of following target devices: μPD750064, 750066, 750068, 75P0076, 750104, 750106, 750108, 75P0116, 753012A, 753016A, 753017A, 75P3018A, 753036, 75P3036, 753204, 753206, 753208, 75P3216, 753304[Note], 754202, 754144, 754244, 754264, 75F4264[Note], 754302, 754304, 75P4308
Throughout	Device under development → Developed: μPD750004, 750006, 750008, 75P0016, 753012, 753016, 753017, 75P3018, 753104, 753106, 753108, 75P3116
Throughout	Deletion of following devices: μPD75402, 75P402
Throughout	Change of identifier to symbol
p.8	Change of description in **2.2.2 Symbol**
p.8	Addition of description to ***2.2.4 Expression***
p.58-60	Addition of **3.3 Register Specification**
p.82	Correction of descriptions in **[Purpose]** and **[Explanation] <2>** in **3.4 (10) continue**
p.86	Addition of **3.4 (12) forever**
p.95-100	Addition of **CHAPTER 5 CONTROL INSTRUCTION**
p.101	Addition of **6.3 Device File**
p.110-124	Addition of type of structured assembler option: **7.4.3 (9) Mode specification (M option)** **(10) Symbol name length specification (S and NS options)** **(11) Debug information output specification (GS and NGS options)** **(12) Device file search path specification (Y option)**
p.125-130	Addition of **7.5 Setting Option from Project Manager**
p.153-167	Addition of item of "Page" to tables in **APPENDIX C LIST OF STATEMENTS OF STRUCTURED ASSEMBLER** through **APPENDIX G OPTION LIST**

Note Under development

A star (★) in the margin indicates the major revisions in this edition

[MEMO]

PREFACE

This manual explains the features, use, and operation of the structured assembler preprocessor program (hereafter referred to as the "structured assembler" or "ST75X") included in the RA75X assembler package (hereafter referred to as the "RA75X").

This manual does not explain programs other than the structured assembler. For these programs, refer to the following manuals:

Manual	RA75X Programs
RA75X Assembler Package User's Manual (Language, Operation)	Assembler Linker Object converter Librarian List converter Library converter
This manual	Structured assembler

[Readers]

This manual is intended for users who understand the functions and instructions of the microcontroller (75X series or 75XL series) under development.

[Target device]

The structured assembler is used for the following devices:

Series Name		Target Device
75X series	Extended high-end	μPD75117H, 75P117H, 75217, 75218, 75P218, 75236, 75237, 75238, 75P238, 75517, 75518, 75P518, 75617A
	High-End	μPD75104, 75104A, 75106, 75108, 75108A, 75P108B, 75108F, 75P108, 75112, 75112F, 75116, 75P116, 75116F, 75116H, 75206, 75208, 75CG208, 75212A, 75216A, 75P216A, 75CG216A, 75336, 75P336, 75352A
	Standard	μPD75004, 75006, 75008, 75P008, 75028, 75036, 75P036, 75048, 75P048, 75064, 75066, 75068, 75P068, 75268, 75304, 75304B, 75306, 75306B, 75308, 75P308, 75308B, 75312, 75312B, 75316, 75P316, 75P316A, 75316B, 75P316B, 75328, 75P328
75XL series		μPD750004, 750006, 750008, 75P0016, 750104, 750106, 750108, 75P0116, 750064, 750066, 750068, 75P0076, 753012, 753016, 753017, 75P3018, 753012A, 753016A, 753017A, 75P3018A, 753036, 75P3036, 753104, 753106, 753108, 75P3116, 753204, 753206, 75P3208, 75P3216, 753304[Note], 754202, 754144, 754244, 754264, 75F4264[Note], 754302, 754304, 75P4308

Note Under development

Caution The structured assembler does not support the low-end devices of the 75X series (μPD75P402, 75402A).

[Organization]

This manual consists of the following chapters:

CHAPTER 1 OVERVIEW

Explains the role, features, and outline of the structured assembler.

CHAPTER 2 WRITING SOURCE PROGRAMS

Explains the general rule for writing a source program, such as the organization of the source program, syntax, and arithmetic operators.

CHAPTER 3 CONTROL STATEMENTS

Explains the features and use of control statements for the structured assembler.

CHAPTER 4 PSEUDOINSTRUCTIONS

Explains the pseudoinstructions of the structured assembler, giving examples.

★ **CHAPTER 5 CONTROL INSTRUCTIONS**

Explains the features and use of control instructions of the structured assembler.

CHAPTER 6 PRODUCT OUTLINE

Explains the related files and operating environment of the structured assembler.

CHAPTER 7 OPERATION

Explains the operation and options of the structured assembler.

CHAPTER 8 I/O FILES

Explains the format of the lists output by the structured assembler.

CHAPTER 9 HOW TO USE PRODUCT

Introduces a method for using the structured assembler effectively.

CHAPTER 10 ERROR MESSAGES AND TERMINATION PROCESSING INFORMATION

Lists the error messages of the structured assembler.

★ **APPENDIXES**

Provide lists of the maximum performance, target products, structured assembler statements, control statements, pseudoinstructions, control instructions, and options.

[Legend]

- ... : Repetition of the same format
- [ ] : May be omitted
- ' ' : Character or character string enclosed by ' '
- ( ) : Character string enclosed by ( )
- Bold : Characters
- __ : Character string to be input, such as a command
- : : Abbreviated portion of program
- CR : Carriage return
- LF : Line feed
- / : Delimiter
- α : Symbol name or register name
- β : Symbol name or register name
- γ : Symbol name or register name
- △ : One or more blank or HT (tab)

[Related Documents]

In addition to this manual, also refer to the following documents:

Document Name	Document Number
RA75X Assembler Package V.5.xx User's Manual - Language	U12385E
RA75X Assembler Package V.5.xx User's Manual - Operation	U12622E
75X Series Structured Assembler Preprocessor Application Note	EEA-1203

[MEMO]

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 OVERVIEW

1.1 Outline of Structured Assembler

The structured assembler (ST75X) is a program used to develop application systems for 75X series or 75XL series microcontrollers and is included in the "RA75X assembler package".

1.1.1 Role of structured assembler

The structured assembler translates programs written in a structured assembly language providing "if ~ else ~ endif" statements and "for ~ next" statements, which allow the programmer to give programs a clear structure, into a source program for an assembler.

Using the structured assembler provides the following three benefits:

(1) Programs are easy to write.
- No need to think about label names for branches.
- Transfer instructions that normally require a lot of coding can be described with symbols.
- Programs can be written by learning a minimum number of mnemonics (excellent transplantability).

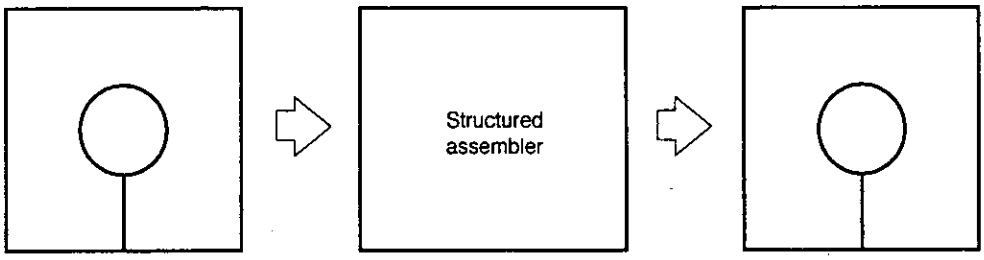
(2) Programs are easy to read.
- Program structure is clear.
- Operation and transfer between memory and registers can be written in one statement.
- Programs developed by others are easy to read.
- Programs are easy to maintain (modify).

(3) Easy desk-top debugging.

1.2 Outline of Features of ST75X

The ST75X analyzes control statements, expressions, pseudoinstructions, and control instructions in a structured assembler source program written in a structured assembler language, and outputs an assembler source program that is used as the input source file of an assembler.

Figure 1-1. Flow of Structured Assembler



Source program written in structured assembly language

Source program written in assembly language

Structured statements are output as comment statements in a secondary source file, in addition to the assembler instructions after conversion and normal assembly language.

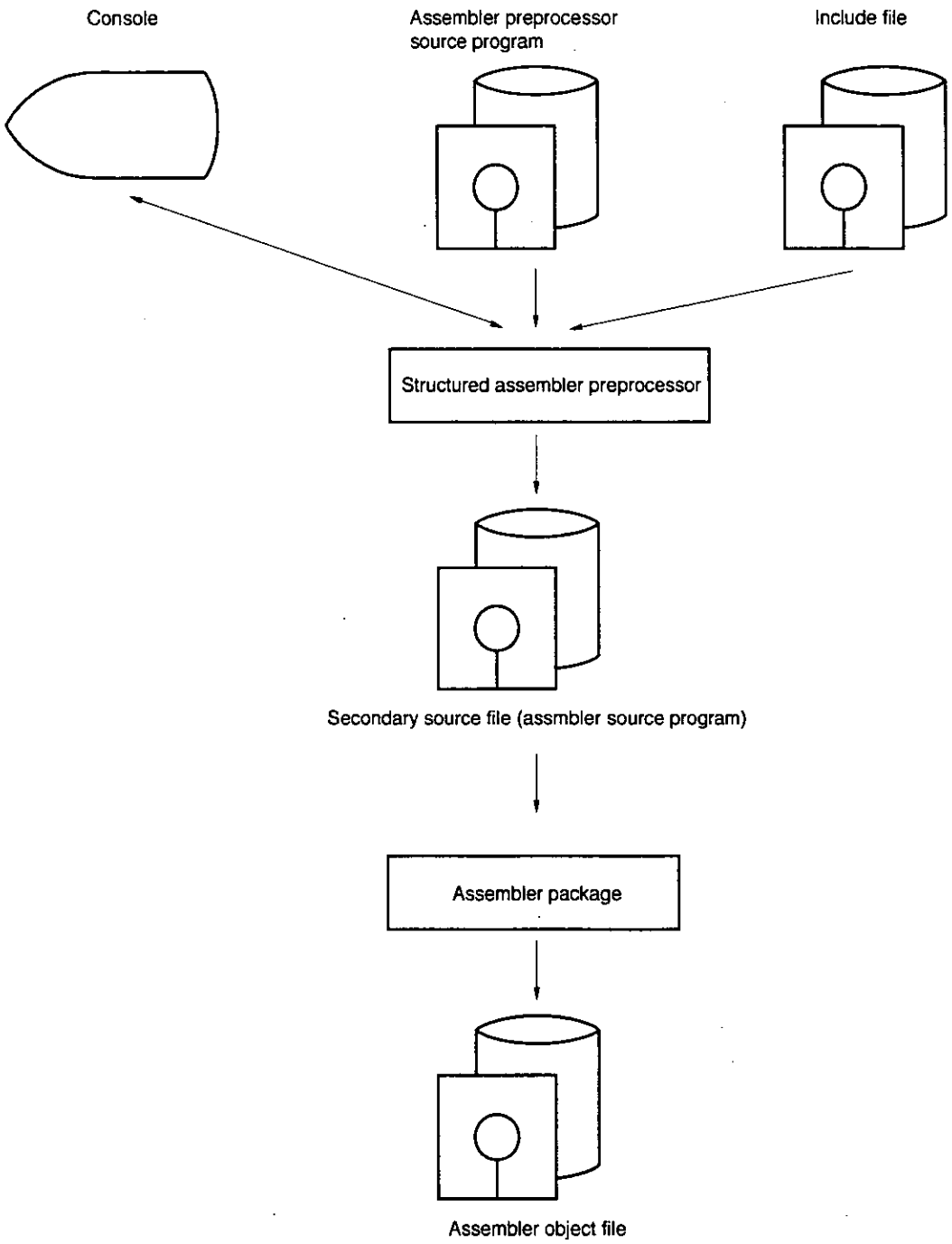
If errors are found, error messages are output.

The features of the ST75X are as follows:

- Many C-like control structures make programming easy.
- C-like assignment statements and assignment operators.
- Control structures and assignment statement for bit processing.
- C-like symbol definition pseudoinstructions, conditional processing features, and include pseudoinstructions available.
- Being a preprocessor that outputs a source program of the assembler, the structured assembler can optimize code after conversion.
- Pseudoinstructions that can be translated to GETI instructions available, giving optimum code output.
- Easy-to-read assembly lists can be created by changing the output position of the assembler source program.

Figure 1-2 shows the flow of program development.

Figure 1-2. Flow of Development



1.3 Before Developing Programs

Bear in mind the following points before starting the development of a program:

1.3.1 Maximum performance

The following limitations apply to source programs for the structured assembler:

Item	Maximum Performance
Length of one line (excluding LF and CR)	254 characters
Number of registered symbols (excluding reserved words)	512
Nesting level of control statements	31 levels
Nesting level of conditional processing instructions	8 levels
#defgeti pseudoinstructions	48
Nesting level of #include pseudoinstructions	1 level
Number of operands sequentially assigned	33

1.3.2 Notes

(1) Label

When defining a label (an assembler symbol indicating an address), write only the definition of the label on one line.

Example

<Correct>

```
SYMBOL:
        NOP
```

<Incorrect>

```
SYMBOL:  NOP
```

(2) Delimiter

When using a blank or horizontal tab as a delimiter in an assembler expression, enclose the expression in parentheses '( )'.

Example 1. In assembly language

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

2. In source program of structured assembler

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

(3) Target device

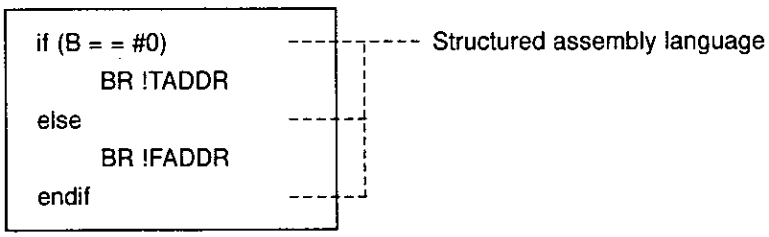
The structured assembler does not support the low-end devices of the 75X series (μPD75P402 and 75402A).

2.1 Basic Structure of Source Programs

2.1.1 Structure of statements

Source programs are written in structured assembly language and assembly language.

Example



(1) Structured assembly language

The structured assembly language has the types of statements listed in Table 2-1 Structured Assembly Language Statements.

Table 2-1. Structured Assembly Language Statements

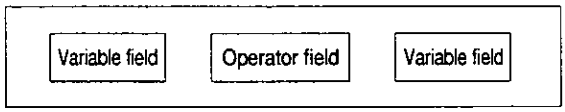
	Type
Expression statements	Assignment operation expression, increment operation expression, decrement operation expression, exchange operation expression, compare operation expression, logical operation expression
Structured statements	if statement, if_bit statement, switch statement, for statement, while statement, while_bit statement, repeat statement, repeat_bit statement, break statement, continue statement, goto statements
Comment statements	—

The compare operation and logical operation expressions cannot be used alone, and are always written with a control statement (for details, refer to **3.4 Purpose of Control Statements**).

(a) Expression statements

An expression consists of the three fields shown in Figure 2-1 Fields of Expression Statements.

Figure 2-1. Field of Expression Statements



- A blank may be inserted between the variable field and operator field.
- The statement can be written starting from any column.
- Assignment operators, increment/decrement operators, and exchange operators can be written in the operator field.
- Register names and immediate data can be written in the variable field (for details, refer to **2.4 Expressions and Operators**).

(b) Control statements

- Control statements include if, if_bit, switch, for, while, while_bit, repeat, repeat_bit, break, continue, and goto statements. For details, refer to **CHAPTER 3 CONTROL STATEMENTS**.

(c) Comment statements

- Character strings following a semicolon ";" are regarded as comment statements. The comment statement is output to the secondary source file as is, without being processed in any way.

(2) Assembly language

For an explanation of the assembly language, refer to **RX75X Assembler Package User's Manual**.
Up to 220 characters can be written on one line (between LF and LF).

2.2 Source Program Format

2.2.1 Character set

To write statements, use alphabetic characters, characters associated with the alphabet, numerals, and special characters.

(1) Alphabet

```
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
```

★ #### (2) Characters associated with the alphabet

```
? _
```

(3) Numerals

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

(4) Special characters

Character	Name	Usage
	Blank	Delimiter between each character
@	Unit price symbol	Indirect addressing start symbol
'	Single quotation marks	Indicate beginning and end of character constant, or specify extended register
"	Double quotation marks	Used to specify disk type file name in #INCLUDE pseudoinstruction
,	Comma	Delimiter between operands
.	Pericd	Bit position symbol of bit symbol
+	Plus	Positive sign or addition operator
–	Minus	Negative sign or subtraction operator
*	Asterisk	Multiplication operator
/	Slash	Division operand
&	Ampersand	Logical product operator
|	Separation symbol	Logical sum operator
^	Up arrow	Exclusive logical sum operator
(	Left parenthesis	Changes operation sequence, or expression of control statement
)	Right parenthesis	
=	Equal	Assignment operator or compare operator
;	Semicolon	Starts comment, or delimits expression in control statement
:	Colon	Delimiter of label
#	Number symbol	First character of pseudoinstruction of structured assembler, or immediate indication symbol
!	Exclamation	Direct addressing start symbol, negative indication symbol
<	Unequal	Compare operators
>	Unequal	
\	Back Slash	Directory specification symbol
$	Dollar	Value of location counter, control instruction indication symbol
HL	Horizontal tab	Blank-equivalent character
LF	Line feed	End of line

★ 2.2.2 Symbol

A symbol is a name given to numeric data or an address.

By using symbols, the contents of the source program are made easier to understand.

The ST75X provides two types of symbols: reserved symbols and user symbols.

(1) Reserved symbols

A reserved symbol is a symbol defined in advance in the ST75X, and must not be defined by the user.

The register names and specific address symbols which are reserved words in the assembly language are recognized as reserved symbols.

Reserved words can be written in upper-case or lower-case characters, or in both upper-case and lower-case characters together.

With the ST75X, specific address symbols are classified into bit, nibble, and byte types by data size.

Specific address symbols are checked to see if they can be read or written. If a specific address symbol is described as "xxx.bit", however, the specific address symbol is checked to see if it can be read or written, but whether the individual bits of the specific address symbol can be read or written is not checked.

For more information on the register names and specific address symbols of the reserved words, refer to **RA75X Assembler Package User's Manual** and **"Before Using Device File"** supplied with the device file.

(2) User symbols

Character strings other than reserved symbols are treated as user symbols.

A user symbol can consist of alphanumeric characters and alphabet-associated characters. Note, however, that user symbols must not start with a numeral.

2.2.3 Constants

★ The structured assembly language does not support constants. Therefore, constants in assembly language are treated as user symbols (for more information on constants of the assembly language, refer to **RA75X Assembler Package User's Manual**).

2.2.4 Expression

An expression consists of constants, special characters, and symbols combined by operators (for the expression of the assembly language, refer to **RA75X Assembler Package User's Manual**).

When the blank or horizontal tab are used as delimiters in the assembly language, enclose the expression in ( ).

Example 1. In assembly language

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

2. In source program of structured assembler

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

★ "++" and "– –" are used for increment and decrement with the ST75X (refer to **Examples 1** and **2** below). When writing "+" and "+" sign, or "–" and "–" sign as the arithmetic operators of the assembler, write them as shown in Examples 3 and 4 below.

Example 1. SYM2 EQU A1– –A2
2. XA = A3++A4
3. SYM2 EQU A1△–△–A2
4. XA = (A3△+△+A4)

★ In an expression in assembly language, compare operators which are identical to the operators of the structured assembly language must not be used because they are processed as the operators of the structured assembly language.

- Assembly language compare operators that can be used
  EQ, NE, GT, GE, LT, LE
- Assembly language compare operators that must not be used
  =, < >, >, >=, <, <=

2.3 Reserved Words

★ 2.3.1 List of reserved words of structured assembly language

Control statements	if, if_bit, elseif, elseif_bit, endif
	switch, case, default, ends
	for, next
	while, while_bit, endw
	repeat, until, until_bit
	forever
	break, continue, goto
Pseudoinstructions	# define
	# ifdef, # else, # endif
	# include
	# defgeti, # endgeti
Operators	+ +, − −, + =, − =, & =, | -, ^ =
	= =, ! =, <, > =, >, < =
	+, −, *, /, &, |, ^
Register flags	CY, A, X, B, C, D, E, L, H, @H, @L,
	XA, BC, DE, HL, DL, XA', BC', DE', HL'
	@DL, @HL+, @HL−, @PCXA, @PCDE
Assembler control instructions	INCLUDE, IC
	TITLE, TT
	LIST, LI, NOLIST, NOLI
	EJECT, EJ
	PROCESSOR, PC
	MODE, MD
	ERRLOG, EL
	MSGLOG, ML
	IFCHR, IFSTR
	LODM, LM
	DEBUGA, DA, DEBUG, DG, NODEBUGA, NODA, NODEBUG, NODB
	SYMBOLS, SB, NOSYMBOLS, NOSB
	XREF, XF, NOXREF, NOXR
	PAGELENGTH, PL, PAGEWIDTH, PWTAB, TAB, TB
	CAP, CA, NOCAP, NOCA
	SYMLEN, SL, NOSYMLEN, NOSL
	GENERATE, GEN, NOGENERATE, NOGEN
	CONDITION, COND, NOCONDITION, NOCOND
	IF, IFDEF, ELSE, ENDIF
	SWITCH, CASE, DEFAULT, ENDS
	DGL, DGS, TOL_INF

2.4 Expressions and Operators

An expression consists of constants (numeric constants and character constants), special characters, and symbols combined using operators. The elements that constitute an expression, except the operators, are called terms, and are called the first term, second term, and so on, from the left in the sequence in which they are written.

The types of operators listed in Table 2-2 Types of Operators are available.

Table 2-2. Types of Operator

Type of Operator	Operator
Assignment operator	=, + =, − =, & =, | =, ^ =
Increment/decrement operator	+ +, − −
Exchange operator	< − >
Compare operator	= =, ! =, <, >, > =, < =
Logical operator	& &, | |

In addition, the types of expressions shown in Table 2-3 Types of Expressions are available.

Table 2-3. Types of Expressions

Type of Expression	Operator Constituting Expression
Assignment expression	Assignment operator
Increment/decrement expression	Increment/decrement operator
Exchange expression	Exchange operator
Compare expression	Compare operator
Logical expression	Logical operator

The meanings of the symbols used in 2.4.1 through 2.4.6 and 3.4 are as follows:

Symbol	Description
reg	X, A, B, C, D, E, H, L
reg1	X, B, C, D, E, H, L
rp1	XA, BC, DE, HL
rp2	BC, DE, HL
rp'	XA, BC, DE, HL, XA', BC', DE', HL'
rp'1	BC, DE, HL, XA', BC', DE', HL'
rpa1	DE, DL
n4	4-bit immediate data or label
n8	8-bit immediate data or label
mem	8-bit immediate data or label[Note]
bit	2-bit immediate data or label
fmem	Immediate data FB0H through FBFH, FF0H through FFFH, or label
pmem	Immediate data FC0H through FFFH or label

Note Only even addresses can be used for 8-bit data processing.

★

Symbol		Target Device	
H	H-II	μPD750004, 750006, 750008, 75P0016, 750104, 750106, 750108, 75P0116, 750064, 750066, 750068, 75P0076, 753012, 753016, 753017, 75P3018, 753012A, 753016A, 753017A, 75P3018A, 753036, 75P3036, 753104, 753106, 753108, 75P3116, 753204, 753206, 75P3208, 75P3216, 753304[Note], 754202, 754144, 754244, 754264, 75F4264[Note], 754302, 754304, 75P4308	75XL series
		μPD75117H, 75P117H, 75217, 75218, 75P218, 75236, 75237, 75238, 75P238, 75517, 75518, 75P518, 75617A	75X series
	H-I	μPD75104, 75104A, 75106, 75108, 75108A, 75P108B, 75108F, 75P108, 75112, 75112F, 75116, 75P116, 75116F, 75116H, 75206, 75208, 75CG208, 75212A, 75216A, 75P216A, 75CG216A, 75336, 75P336, 75352A	
S		μPD75004, 75006, 75008, 75P008, 75028, 75036, 75P036, 75048, 75P048, 75064, 75066, 75068, 75P068, 75268, 75304, 75304B, 75306, 75306B, 75308, 75P308, 75308B, 75312, 75312B, 75316, 75P316, 75P316A, 75316B, 75P316B, 75328, 75P328	

Note Under development

Caution The structured assembler does not support the low-end devices of the 75X series (μPD75P402 and 75402A).

Remark H-I : High-End I
HI-II: High-End II
S : Standard

Symbol	Description
○	Can be used
×	Cannot be used

2.4.1 Assignment operator

An assignment operator is used to assign the second term in an expression to the first term.

Table 2-4 Types of Assignment Operator lists the assignment operators available.

Table 2-4. Types of Assignment Operator

Assignment Operator	Format	Purpose
Assignment (=)	$\alpha = \beta$	$\alpha \leftarrow \beta$
Addition assignment (+=)	α += β	$\alpha \leftarrow \alpha + \beta$
Subtraction assignment (−=)	α −= β	$\alpha \leftarrow \alpha - \beta$
Logical product assignment (&=)	α &= β	$\alpha \leftarrow \alpha$ & β
Logical sum assignment (|=)	α |= β	$\alpha \leftarrow \alpha$ | β
Exclusive logical sum assignment (^=)	α ^= β	$\alpha \leftarrow \alpha$ ^ β

Assignment Operators

(1) Assignment (=)

[Format]

$\alpha = \beta$

[Purpose]

Assigns β to α.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

★

α	β	Operation	Skip Condition	H		S
				H-II	H-I	
A	#n4	A←n4	String-effect A	○	○	○
reg1	#n4	reg1←n4		○	○	○
XA	#n8	XA←n8		○	○	○
HL	#n8	HL←n8	String-effect B	○	○	○
rp2	#n8	rp2←n8		○	○	○
A	@HL	A←(HL)		○	○	○
A	@HL+	A←(HL), then L←L+1	L = 0	○	○	○Note
A	@HL−	A←(HL), then L←L−1	L = FH	○	○	○Note
A	@rpa1	A←(rpa1)		○	○	○
XA	@HL	XA←(HL)		○	○	○
@HL	A	(HL)←A		○	○	○
@HL	XA	(HL)←XA		○	○	○
A	mem	A←(mem)		○	○	○
XA	mem	XA←(mem)		○	○	○
mem	A	(mem)←A		○	○	○
mem	XA	(mem)←XA	String-effect A	○	○	○
A	reg	A←reg		○	○	×
XA	rp'	XA←rp'		○	○	×
reg1	A	reg1←A		○	○	○
rp'1	XA	rp'1←XA		○	○	○
XA	@PCDE	XA←$(PC_{13-8}+DE)_{ROM}$		○	○	○
XA	@PCXA	XA←$(PC_{13-8}+XA)_{ROM}$		○	○	○
CY	fmem.bit	CY←(fmem.bit)		○	○	×
CY	pmem.@L	CY←(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$))		○	○	×
CY	@H+mem.bit	CY←(H+mem$_{3-0}$.bit)		○	○	×
fmem.bit	CY	(fmem.bit)←CY		○	○	×
pmem.@L	CY	(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))←CY		○	○	×
@H+mem.bit	CY	(H+mem$_{3-0}$.bit)←CY		○	○	×
XA	@BCDE	XA←$(B_{2-0}+CDE)_{ROM}$		○	×	×
XA	@BCXA	XA←$(B_{2-0}+CXA)_{ROM}$		○	×	×

★ **Note** MOV and INCS or DESC instructions are translated in combination. Refer to **[Example]** on the next page.

Assignment Operators

[Generated Instructions]

Comparison is performed in the following priority sequence. If the result matches, the instruction is translated.

<1> If α or β: CY

MOV1 α, β

<2> If β: @PCDE, @PCXA other than <1>

MOVT α, β

<3> Other than <1> and <2>

MOV α, β

★ **[Example]**

<Input source file>

```
CY = @H+PFG.0
XA = @PCXA
MODEP = XA
A = @HL+
```

<Output source file>

```
MOV1    CY,@H+PFG.0  ;CY = @H+PFG.0
MOVT    XA,@PCXA     ;XA = @PCXA
MOV     MODEP,XA     ;MODEP = XA
MOV     A,@HL        ;A = @HL+
INCS    L
```

Remark The output result of "A=@HL+" is only when the target device is "S".

Assignment Operator

(2) Register name specification assignment (=)

[Format]

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

[Purpose]

Assigns β to γ and the contents of γ to α.

[Explanation]

<1> Acceptable formats for α, β, and γ and the corresponding operations are shown below.

	α	β	γ	Operation	H	S
	@HL	#n4	A	(HL)←n4	○	○
Note 1	@HL	@HL+	A	(H (L+1))←(HL), then L←L+1	○	○Note 3
Note 2	@HL	@HL−	A	(H (L−1))←(HL), then L←L−1	○	○Note 3
	@HL	@rpa1	A	(HL)←(rpa1)	○	○
	@HL	mem	A	(HL)←(mem)	○	○
	@HL	reg1	A	(HL)←reg1	○	○
	mem	#n4	A	(mem)←n4	○	○
	mem	@HL	A	(mem)←(HL)	○	○
Note 1	mem	@HL+	A	(mem)←(HL), then L←L+1	○	○Note 3
Note 2	mem	@HL−	A	(mem)←(HL), then L←L−1	○	○Note 3
	mem	@rpa1	A	(mem)←(rpa1)	○	○
	mem	mem	A	(mem)←(mem)	○	○
	mem	reg1	A	(mem)←reg1	○	○
	reg1	#n4	A	reg1←n4	○	○
	reg1	@HL	A	reg1←(HL)	○	○
Note 1	reg1	@HL+	A	reg1←(HL), then L←L+1	○	○Note 3
Note 2	reg1	@HL−	A	reg1←(HL), then L←L−1	○	○Note 3
	reg1	@rpa1	A	reg1←(rpa1)	○	○
	reg1	mem	A	reg1←mem	○	○
	reg1	reg1	A	reg1←reg1	○	○
	@HL	#n8	XA	(HL)←n8	○	○
	@HL	mem	XA	(HL)←(mem)	○	○
	@HL	rp'	XA	(HL)←rp'	○	○
	mem	#n8	XA	(mem)←n8	○	○
	mem	mem	XA	(mem)←(mem)	○	○
	mem	@HL	XA	(mem)←(HL)	○	○
	mem	rp'	XA	(mem)←rp'	○	○
	rp'1	#n8	XA	rp'1←n8	○	○
	rp'1	mem	XA	rp'1←(mem)	○	○
	rp'1	@HL	XA	rp'1←(HL)	○	○
	rp'1	rp'	XA	rp'1←rp'	○	○

Notes 1. If L = 0, assignment is not performed correctly.

2. If L = FH, assignment is not performed correctly.

★ 3. MOV and INCS or DESC instructions are translated in combination. Refer to **[Example]** on the next page.

Assignment Operator

α	β	γ	Operation	H	S
fmem.bit	fmem.bit	CY	(fmem.bit)←(fmem.bit)	○	×
fmem.bit	pmem.@L	CY	(fmem.bit)←(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))	○	×
fmem.bit	@H+mem.bit	CY	(fmem.bit)←(H+mem$_{3-0}$.bit)	○	×
pmem.@L	fmem.bit	CY	(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))←(fmem.bit)	○	×
pmem.@L	pmem.@L	CY	(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))←(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))	○	×
pmem.@L	@H+mem.bit	CY	(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))←(H+mem$_{3-0}$.bit)	○	×
@H+mem.bit	fmem.bit	CY	(H+mem$_{3-0}$.bit)←(fmem.bit)	○	×
@H+mem.bit	pmem.@L	CY	(H+mem$_{3-0}$.bit)←(pmem$_{7-2}$+L$_{3-2}$.bit(L$_{1-0}$))	○	×
@H+mem.bit	@H+mem.bit	CY	(H+mem$_{3-0}$.bit)←(H+mem$_{3-0}$.bit)	○	×

<2> Note that specified γ is changed to the contents of β.

<3> A blank is necessary between β and (γ).

[Generated Instructions]

<1> If γ: CY

```
MOV1    CY, β
MOV1    α, CY
```

<2> If γ: register

```
MOV     γ, β
MOV     α, γ
```

★ **[Example]**

<Input source program>

```
TMOD = #0AH (XA)
@HL = @HL + (A)
```

<Output source program>

```
MOV    XA,#0AH ;TMOD = #0AH (XA)
MOV    TMOD,XA
MOV    A,@HL    ;@HL = @HL+ (A)
INCS   L
MOV    @HL,A
```

Remark The output result of "@HL = @HL+ (A)" is only when the target device is "S".

Assignment Operator

(3) Addition assignment (+=)

[Format]

α += β, CY [$\triangle$ (γ)] γ: register name

[Purpose]

Adds two terms, α and β ($\alpha + \beta$), and assigns the result to α.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

α	β	CY	Operation	Skip Condition	H	S
A	#n4	—	A←A+n4	carry	○	○
XA	#n8	—	XA←XA+n8	carry	○	×
A	@HL	—	A←A+(HL)	carry	○	○
XA	rp'	—	XA←XA+rp'	carry	○	×
rp'1	XA	—	rp'1←rp'1+XA	carry	○	×
A	@HL	Occurs	A, CY←A+(HL)+CY	—	○	○
XA	rp'	Occurs	XA, CY←XA+rp'+CY	—	○	×
rp'1	XA	Occurs	rp'1, CY←rp'1+XA+CY	—	○	×

Remark Write the assembly language as is for addition with carry.

★ **[Generated Instructions]**

<1> When register is not specified

(a) If α+=β

```
ADDS    α, β
```

(b) If α+=β, CY

```
ADDC    α, β
```

<2> When register is specified

(a) If α+=β(γ)

```
MOV     γ, α
ADDS    γ, β
MOV     α, γ
```

(b) If α+=β, CY (γ)

```
MOV     γ, α
ADDC    γ, β
MOV     α, γ
```

Assignment Operator

★ **[Example]**

<Input source program>

```
XA += #0CH
A += @HL,CY
X += @HL,CY (A)
```

<Output source program>

```
ADDS    XA,#0CH  ;XA += #0CH
ADDC    A,@HL    ;A += @HL,CY
MOV     A,X      ;X += @HL,CY (A)
ADDC    A,@HL
MOV     X,A
```

Assignment Operator

(4) Subtraction assignment (–=)

[Format]

α –= β, CY △ [γ] γ: register name

[Purpose]

Subtracts second term, β from α ($\alpha - \beta$), and assigns the result to α.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

α	β	CY	Operation	Skip Condition	H	S
A	@HL	—	A←A–(HL)	borrow	○	○
XA	rp'	—	XA←XA–rp'	borrow	○	×
rp'1	XA	—	rp'1←rp'1–XA	borrow	○	×
A	@HL	Occurs	A, CY←A–(HL)–CY	—	○	○
XA	rp'	Occurs	XA, CY←XA–rp'–CY	—	○	×
rp'1	XA	Occurs	rp'1, CY←rp'1–XA–CY	—	○	×

Remark Write the assembly language as is for subtraction with carry.

★ **[Generated Instructions]**

<1> When register is not specified

(a) If α–=β

```
SUBS    α, β
```

(b) If α–=β, CY

```
SUBC    α, β
```

<2> When register is specified

(a) If α–=β(γ)

```
MOV     γ, α
SUBS    γ, β
MOV     α, γ
```

(b) If α–=β, CY (γ)

```
MOV     γ, α
SUBC    γ, β
MOV     α, γ
```

Assignment Operator

★ **[Example]**

<Input source program>

```
A -= @HL
A -= @HL,CY
X -= @HL,CY (A)
```

<Output source program>

```
SUBS    A,@HL   ;A -= @HL
SUBC    A,@HL   ;A -= @HL,CY
MOV     A,X     ;X -= @HL,CY (A)
SUBC    A,@HL
MOV     X,A
```

Assignment Operator

(5) Logical product assignment (&=)

[Format]

α &= β [△ (γ)] γ: register name or CY

[Purpose]

ANDs the bits of two terms, α and β ($\alpha \cap \beta$), and assigns the result to α.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

α	β	Operation	H	S
A	#n4	A←A∩n4	○	○
A	@HL	A←A∩(HL)	○	○
XA	rp'	XA←XA∩rp'	○	×
rp'1	XA	rp'1←rp'1∩XA	○	×
CY	fmem.bit	CY←CY∩(fmem.bit)	○	○
CY	pmem.@L	CY←CY∩(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$))	○	○
CY	@H+mem.bit	CY←CY∩(H+mem$_{3-0}$.bit)	○	Note

Note This cannot be used with the μPD75048 when MBS = 4, 5, 6, and 7.

★ **Caution When using a specific address symbol as β, an error occurs if the specific address symbol cannot be read.**

[Generated Instructions]

<1> When register is not specified (α&=β)

(a) If α: CY

AND1 CY, β

(b) Other than (a)

AND α, β

Assignment Operator

★ <2> When register is specified (α&=β (γ))

(a) If target device is "H" and if γ:CY

```
MOV1    CY, α
AND1    CY, β
MOV1    α, CY
```

(b) If target device is "S" and γ:CY

```
SET1    CY
SKT     α
CLR1    CY
AND1    CY, β
SET1    α
SKT     CY
CLR1    α
```

(c) Other than (a) and (b)

```
MOV     γ, α
AND     γ, β
MOV     α, γ
```

★ **[Example]**

<Input source program>

```
CY &= @H+PFG.0
XA &= BC
PORT2.0 &= PORT2.1 (CY)
B &= #07H (A)
```

<Output source program>

```
AND1    CY,@H+PFG.0     ;CY &= @H+PFG.0
AND     XA,BC           ;XA &= BC
MOV1    CY,PORT2.0      ;PORT2.0 &= PORT2.1 (CY)
AND1    CY,PORT2.1
MOV1    PORT2.0,CY
MOV     A,B             ;B &= #07H (A)
AND     A,#07H
MOV     B,A
```

Assignment Operator

(6) Logical sum assignment (|=)

[Format]

α |= β [Δ (γ)] γ: register name or CY

[Purpose]

ORs the bits of two terms, α and β (α U β), and assigns the result to α.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

α	β	Operation	H	S
A	#n4	A←AUn4	○	○
A	@HL	A←AU(HL)	○	○
XA	rp'	XA←XAUrp'	○	×
rp'1	XA	rp'1←rp'1UXA	○	×
CY	fmem.bit	CY←CYU(fmem.bit)	○	○
CY	pmem.@L	CY←CYU(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$))	○	○
CY	@H+mem.bit	CY←CYU(H+mem$_{3-0}$.bit)	○	Note

Note This cannot be used with the μPD75048 when MBS = 4, 5, 6, and 7.

Caution When using a specific address symbol as β, an error occurs if the specific address symbol cannot be read.

[Generated Instructions]

Comparison is performed in the following priority sequence, and if the result matches, the instruction is translated.

<1> When register is not specified (α|=β)

(a) If α: CY

OR1 CY, β

(b) Other than (a)

OR α, β

Assignment Operator

★ <2> When register is specified (α|=$\beta(\gamma)$)

(a) If target device is "H" and if γ:CY

```
MOV1    CY, α
OR1     CY, β
MOV1    α, CY
```

(b) If target device is "S" and γ:CY

```
SET1    CY
SKT     α
CLR1    CY
OR1     CY, β
SET1    α
SKT     CY
CLR1    α
```

(c) Other than (a) and (b)

```
MOV     γ, α
OR      γ, β
MOV     α, γ
```

★ **[Example]**

<Input source program>

```
CY |= @H+PFG.0
XA |= BC
PCRT2.0 |= PORT2.1 (CY)
B |= #07H (A)
```

<Output source program>

```
OR1     CY,@H+PFG.0  ;CY |= @H+PFG.0
OR      XA,BC        ;XA |= BC
MCV1    CY,PORT2.0   ;PORT2.0 |= PORT2.1 (CY)
OR1     CY,PORT2.1
MCV1    PORT2.0,CY
MCV     A,B          ;B |= #07H (A)
OR      A,#07H
MCV     B,A
```

Assignment Operator

(7) Logical exclusive sum assignment (^=)

[Format]

α ^= β [$\triangle$ (γ)] γ: register name or CY

[Purpose]

Exclusive-ORs the bits of two terms, α and β (α ^ β), and assigns the result to α.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

α	β	Operation	H	S
A	#n4	A←A ∀ n4	○	○
A	@HL	A←A ∀(HL)	○	○
XA	rp'	XA←XA ∀ rp'	○	×
rp'1	XA	rp'1←rp'1 ∀ XA	○	×
CY	fmem.bit	CY←CY∀(fmem.bit)	○	○
CY	pmem.@L	CY←CY∀(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$))	○	○
CY	@H+mem.bit	CY←CY∀(H+mem$_{3-0}$.bit)	○	Note

Note This cannot be used with the μPD75048 when MBS = 4, 5, 6, and 7.

Caution When using a specific address symbol as β, an error occurs if the specific address symbol cannot be read.

[Generated Instructions]

Comparison is performed in the following priority sequence, and if the result matches, the instruction is translated.

<1> When register is not specified (α^=β)

(a) If α: CY

XOR1 CY, β

(b) Other than (a)

XOR α, β

Assignment Operator

★ <2> When register is specified (α^=$\beta(\gamma)$)

(a) If target device is "H" and if γ:CY

```
MOV1    CY, α
XOR1    CY, β
MOV1    α, CY
```

(b) If target device is "S" and γ:CY

```
SET1    CY
SKT     α
CLR1    CY
XOR1    CY, β
SET1    α
SKT     CY
CLR1    α
```

(c) Other than (a) and (b)

```
MOV     γ, α
XOR     γ, β
MOV     α, γ
```

★ **[Example]**

<Input source program>

```
CY ^= @H+PFG.0
XA ^= BC
PORT2.0 ^= PORT2.1 (CY)
B ^= #07H (A)
```

<Output source program>

```
XOR1    CY,@H+PFG.0     ;CY ^= @H+PFG.0
XOR     XA,BC           ;XA ^= BC
MOV1    CY,PORT2.0      ;PORT2.0 ^= PORT2.1 (CY)
XOR1    CY,PORT2.1
MOV1    PORT2.0,CY
MOV     A,B             ;B ^= #07H (A)
XOR     A,#07H
MOV     B,A
```

Assignment Operator

(8) Bit set

[Format]

α = (β) ... =1 (γ) γ: register name

[Purpose]

Sets a bit of each term.

Up to 32 terms can be written on one line.

[Explanation]

The operations that can be performed are shown below.

α	Operation	H	S
mem.bit	(mem.bit)←1	○	○
fmem.bit	(fmem.bit)←1	○	○
pmem.@L	(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$))←1	○	○
@H+mem.bit	(@H+mem$_{3-0}$.bit)←1	○	○
CY	CY←1	○	○

[Generated Instructions]

Comparison is performed in the following priority sequence, and if the result matches, the instruction is translated.

<1> If α: CY

SET1 CY

<2> Other than <1>

SET1 α

[Example]

<Input source program>

```
CY = 1
IRQBT = 1
IRQBT = TFLG.1 = @H+PFG.2 = 1 (CY)
```

<Output source program>

```
SET1    CY          ;CY = 1
SET1    IRQBT       ;IRQBT = 1
SET1    CY          ;IRQBT = TFLG.1 = @H+PFG.2 = 1 (CY)
SET1    @H+PFG.2
SET1    TFLG.1
SET1    IRQBT
```

Assignment Operator

(9) Bit clear

[Format]

$\alpha = (\beta) \ldots = 0\ (\gamma)$ γ: register name

[Purpose]

Clears the bits of each term.

Up to 32 terms can be written on one line.

[Explanation]

The operations that can be performed are shown below.

α	Operation	H	S
mem.bit	(mem.bit)←0	○	○
fmem.bit	(fmem.bit)←0	○	○
pmem.@L	($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))←0	○	○
@H+mem.bit	(@H+mem_{3-0}.bit)←0	○	○
CY	CY←0	○	○

[Generated Instructions]

Comparison is performed in the following priority sequence, and if the result matches, the instruction is translated.

<1> If α: CY

```
CLR1    CY
```

<2> Other than <1>

```
CLR1    α
```

[Example]

<Input source program>

```
CY = 0
IRQBT = 0
IRQBT = TFLG.1 = @H+PFG.2 = 0 (CY)
```

<Output source program>

```
CLR1    CY      ;CY = 0
CLR1    IRQBT   ;IRQBT = 0
CLR1    CY      ;IRQBT = TFLG.1 = @H+PFG.2 = 0 (CY)
CLR1    @H+PFG.2
CLR1    TFLG.1
CLR1    IRQBT
```

2.4.2 Increment/decrement operator

The increment/decrement operator increments/decrements the value of a term by 1.

The types of increment/decrement operators are shown in Table 2-5 Types of Increment/Decrement Operator.

Table 2-5. Types of Increment/Decrement Operator

Operator	Format	Purpose
Increment operator (++)	α++	$\alpha \leftarrow \alpha+1$
Decrement operator (– –)	α– –	$\alpha \leftarrow \alpha-1$

Increment/Decrement Operator

(1) Increment (++)

[Format]

```
α++
```

[Purpose]

Adds 1 to the contents of α.

[Explanation]

Acceptable formats for α and the corresponding operations are shown below.

α	Operation	Skip Condition	H	S
reg	reg←reg+1	reg = 0	○	○
rp1	rp1←rp1+1	rp1 = 00H	○	×
@HL	(HL)←(HL)+1	(HL) = 0	○	○
mem	(mem)←(mem)+1	(mem) = 0	○	○

★ **Caution When a specific address symbol is used, an error occurs if the specific address symbol cannot be read/written.**

[Generated Instructions]

INCS α

[Example]

<Input source program>

```
HL ++
```

<Output source program>

```
INCS      HL        ; HL ++
```

Increment/Decrement Operator

(2) Decrement (– –)

[Format]

```
α– –
```

[Purpose]

Subtracts 1 from the contents of α.

[Explanation]

Acceptable formats for α and the corresponding operations are shown below.

α	Operation	Skip Condition	H	S
reg	reg←reg–1	reg = FH	○	○
rp'	rp'1←rp'1–1	rp = FFH	○	×

[Generated Instructions]

DECS α

[Example]

<Input source program>

```
HL --
```

<Output source program>

```
DECS      HL          ; HL --
```

2.4.3 Exchange operator

The exchange operator exchanges the values of the first term and second term.

The exchange operator shown in Table 2-6 Exchange Operator is available.

Table 2-6. Exchange Operator

Operator	Format	Purpose
Exchange operator (<->)	α <-> β	$\alpha \leftarrow \alpha$ <-> β

Exchange Operator

(1) Exchange (<->)

[Format]

α <-> β [$\triangle$ (γ)] γ: Register name

[Purpose]

Exchanges the contents of first term α and second term β.

[Explanation]

Acceptable formats for α and β and the corresponding operations are shown below.

α	β	Operation	Skip Condition	H	S
A	@HL	A↔(HL)		○	○
A	@HL+	A↔(HL), then L←L+1	L = 0	○	○Note
A	@HL−	A↔(HL), then L←L−1	L = F	○	○Note
A	@rpa1	A↔(rpa1)		○	○
XA	@HL	XA↔(HL)		○	○
A	mem	A↔(mem)		○	○
XA	mem	XA↔(mem)		○	○
A	reg1	A↔reg1		○	○
XA	rp'	XA↔rp'		○	○

★ **Note** XCH and INCS or DECS instructions are translated in combination.

[Generated Instructions]

<1> When register is not specified (α <-> β)

```
XCH     α, β
```

<2> When register is specified (α <-> β (γ))

```
MOV     γ, α
XCH     γ, β
MOV     α, γ
```

★ **[Example]**

<Input source program>

```
A <-> @HL+
DATA1 <-> DATA2 (XA)
```

<Output source program>

```
XCH     A,@HL   ;A <-> @HL+
INCS    L
MOV     XA,DATA1;DATA1 <-> DATA2 (XA)
XCH     XA,DATA2
MOV     DATA1,XA
```

2.4.4 Compare operator

A compare operator compares the values of the first term and second term and determines whether the result of the comparison is true or false.

A compare operator, the first term, and the second term constitute a compare expression.

A compare expression can be used in a control statement as a conditional expression, but cannot be used alone.

Table 2-7 Types of Compare Operator lists the compare operators available.

Table 2-7. Types of Compare Operator

Operator	Format	Purpose
==	α == β	True if $\alpha = \beta$, false if $\alpha \neq \beta$
!=	α != β	True if $\alpha \neq \beta$, false if $\alpha = \beta$
<	$\alpha < \beta$	True if $\alpha < \beta$, false if $\alpha >= \beta$
>	$\alpha > \beta$	True if $\alpha > \beta$, false if $\alpha <= \beta$
>=	$\alpha >= \beta$	True if $\alpha >= \beta$, false if $\alpha < \beta$
<=	$\alpha <= \beta$	True if $\alpha <= \beta$, false if $\alpha > \beta$

Compare Operator

(1) ==

[Format]

α == β [$\triangle$ (γ)] γ: register name

[Purpose]

<1> True if the contents of the two terms, α and β, are equal; otherwise, false.

<2> If γ is specified, the contents of α are transferred to γ. True if γ and the contents of β are equal; otherwise, false.

[Explanation]

Acceptable formats for α, β, and γ and the cases where the result is true and false are shown below.

★

	α	β	γ	True	False	H	S
	reg	#n4	—	reg = n4	reg≠n4	○	○
	@HL	#n4	—	(HL = n4	(HL) ≠ n4	○	○
	@HL	#n4	A	(HL) = n4	(HL) ≠ n4	○	○
	#n4	#n4	A	n4 = n4	n4 ≠ n4	○	○
	#n4	#n4	reg1	n4 = n4	n4 ≠ n4	○	○
Note 1	@HL+	#n4	A	(HL) = n4	(HL) ≠ n4	○	○
Note 2	@HL−	#n4	A	(HL) = n4	(HL) ≠ n4	○	○
	@rpa1	#n4	A	(rpa1) = n4	(rpa1) ≠ n4	○	○
	mem	#n4	A	(mem) = n4	(mem) ≠ n4	○	○
	A	#n4	@HL	A = n4	A ≠ n4	○	○
	A	@HL	—	A = (HL)	A ≠ (HL)	○	○
	reg1	@HL	A	reg1 = (HL)	reg1 ≠ (HL)	○	○
	XA	@HL	—	XA = (HL)	XA ≠ (HL)	○	×
	rp'	@HL	XA	rp' = (HL)	rp' ≠ (HL)	○	×
	#n4	@HL	A	n4 = (HL)	n4 ≠ (HL)	○	○
	@rpa1	@HL	A	(rpa1) = (HL)	(rpa1) ≠ (HL)	○	○
	mem	@HL	A	(mem) = (HL)	(mem) ≠ (HL)	○	○
	mem	@HL	XA	(mem) = (HL)	(mem) ≠ (HL)	○	×
	#n8	@HL	XA	n8 = (HL)	n8 ≠ (HL)	○	×
	A	reg	—	A = reg	A ≠ reg	○	○
	reg1	reg	A	reg1 = reg	reg1 ≠ reg	○	○
	#n4	reg	A	n4 = reg	n4 ≠ reg	○	○
	@HL	reg	A	(HL) = reg	(HL) ≠ reg	○	○
Note 1	@HL+	reg	A	(HL) = reg	(HL) ≠ reg	○	○
Note 2	@HL−	reg	A	(HL) = reg	(HL) ≠ reg	○	○
	@rpa1	reg	A	(rpa1) = reg	(rpa1) ≠ reg	○	○
	mem	reg	A	(mem) = reg	(mem) ≠ reg	○	○
	XA	rp'	—	XA = rp'	XA ≠ rp'	○	×
	rp'	rp'	XA	rp' = rp'	rp' ≠ rp'	○	×
	#n8	rp'	XA	n8 = rp'	n8 ≠ rp'	○	×
	mem	rp'	XA	(mem) = rp'	(mem) ≠ rp'	○	×
	@HL	rp'	XA	(HL) = rp'	(HL) ≠ rp'	○	×

Notes 1. The value of the L register is incremented by 1.
Note that comparison is not performed if L = FH.

2. The value of L register is decremented by 1.
Note that comparison is not performed if L = 0H.

Compare Operator

[Generated Instructions]

<1> When register is not specified (α==β)

```
SKE     α, β
BR      ?LFALSE
```

<2> When register is specified (α==β(γ))

```
MOV     γ, α
SKE     γ, β
BR      ?LFALSE
```

[Example]

<Input source program>

```
IF (ABC == #5) (A)
        CALL    !XXX
ELSE
        CALL    !YYY
ENDIF
```

<Output source program>

```
                MOV     A,ABC     ;IF (ABC == #5) (A)
                SKE     A,#5
                BR      ?L1
                CALL    !XXX      ;       CALL    !XXX
                BR      ?L2
?L1:                              ;ELSE
                CALL    !YYY      ;       CALL    !YYY
?L2:                              ;ENDIF
```

Compare Operator

(2) !=

[Format]

α != β [$\triangle$ (γ)] γ: register name

[Purpose]

<1> True if the contents of two terms, α and β, are equal; otherwise, false.

<2> If γ is specified, the contents of α are transferred to γ. True if γ and the contents of β are equal; otherwise, false.

Remark The object effect is better if "==" is used instead of "!=".

★ **[Explanation]**

Acceptable formats for α, β, and γ and the cases where the result is true and false are shown below.

	α	β	γ	True	False	H	S
	reg	#n4	—	reg ≠ n4	reg = n4	○	○
	@HL	#n4	—	(HL) ≠ n4	(HL) = n4	○	○
	@HL	#n4	A	(HL) ≠ n4	(HL) = n4	○	○
	#n4	#n4	A	n4 ≠ n4	n4 = n4	○	○
	#n4	#n4	reg1	n4 ≠ n4	n4 = n4	○	○
Note 1	@HL+	#n4	A	(HL) ≠ n4	(HL) = n4	○	○
Note 2	@HL–	#n4	A	(HL) ≠ n4	(HL) = n4	○	○
	@rpa1	#n4	A	(rpa1) ≠ n4	(rpa1) = n4	○	○
	mem	#n4	A	(mem) ≠ n4	(mem) = n4	○	○
	A	#n4	@HL	A ≠ n4	A = n4	○	○
	A	@HL	—	A ≠ (HL)	A = (HL)	○	○
	reg1	@HL	A	reg1 ≠ (HL)	reg1 = (HL)	○	○
	XA	@HL	—	XA ≠ (HL)	XA = (HL)	○	×
	rp'	@HL	XA	rp' ≠ (HL)	rp' = (HL)	○	×
	#n4	@HL	A	n4 ≠ (HL)	n4 = (HL)	○	○
	@rpa1	@HL	A	(rpa1) ≠ (HL)	(rpa1) = (HL)	○	○
	mem	@HL	A	(mem) ≠ (HL)	(mem) = (HL)	○	○
	mem	@HL	XA	(mem) ≠ (HL)	(mem) = (HL)	○	×
	#n8	@HL	XA	n8 ≠ (HL)	n8 = (HL)	○	×
	A	reg	—	A ≠ reg	A = reg	○	○
	reg1	reg	A	reg1 ≠ reg	reg1 = reg	○	○
	#n4	reg	A	n4 ≠ reg	n4 = reg	○	○
	@HL	reg	A	(HL) ≠ reg	(HL) = reg	○	○
Note 1	@HL+	reg	A	(HL) ≠ reg	(HL) = reg	○	○
Note 2	@HL–	reg	A	(HL) ≠ reg	(HL) = reg	○	○
	@rpa1	reg	A	(rpa1) ≠ reg	(rpa1) = reg	○	○
	mem	reg	A	(mem) ≠ reg	(mem) = reg	○	○
	XA	rp'	—	XA ≠ rp'	XA = rp'	○	×
	rp'	rp'	XA	rp' ≠ rp'	rp' = rp'	○	×
	#n8	rp'	XA	n8 ≠ rp'	n8 = rp'	○	×
	mem	rp'	XA	(mem) ≠ rp'	(mem) = rp'	○	×
	@HL	rp'	XA	(HL) ≠ rp'	(HL) = rp'	○	×

Notes 1. The value of the L register is incremented by 1.
Note that comparison is not performed if L = FH.

2. The value of L register is decremented by 1.
Note that comparison is not performed if L = 0H.

Compare Operator

[Generated Instructions]

<1> When register is not specified (α!=β)

```
        SKE     α, β
        BR      ?LTRUE
        BR      ?LFALSE
?LTRUE:
```

<2> When register is specified (α!=$\beta(\gamma)$)

```
        MOV     γ, α
        SKE     γ, β
        BR      ?LTRUE
        BR      ?LFALSE
?LTRUE:
```

(Some control instructions do not output ?LTRUE:.)

[Example]

<Input source program>

```
if ( A != @HL )
            CALL    !XXX
else
            CALL    !YYY
endif
if ( XYZ != B ) ( A )
            CALL    !PPP
endif
```

<Output source program>

```
                SKE     A,@HL   ;if ( A != @HL )
                BR      ?L2
                BR      ?L1
?L2:
                CALL    !XXX    ;           CALL    !XXX
                BR      ?L3
?L1:                            ;else
                CALL    !YYY    ;           CALL    !YYY
?L3:                            ;endif
                MOV     A,XYZ   ;if ( XYZ != B ) ( A )
                SKE     A,B
                BR      ?L5
                BR      ?L4
?L5:
                CALL    !PPP    ;           CALL    !PPP
?L4:                            ;endif
```

Compare Operator

(3) <

[Format]

$\alpha < \beta$ [$\triangle$ (γ)] γ: register name

[Purpose]

<1> True if the contents of β are greater than those of α; otherwise, false.

<2> If γ is specified, the contents of α are transferred to γ. True if the contents of β are greater than those of γ; otherwise, false.

[Explanation]

<1> Acceptable formats for α, β, and γ and the cases where the result is true and false are shown below.

α	β	γ	True	False	H	S
A	@HL	—	A < (HL)	A >= (HL)	○	○
reg1	@HL	A	reg1 < (HL)	reg1 >= (HL)	○	○
#n4	@HL	A	n4 < (HL)	n4 >= (HL)	○	○
@rpa1	@HL	A	(rpa1) < (HL)	(rpa1) >= (HL)	○	○
mem	@HL	A	(mem) < (HL)	(mem) >= (HL)	○	○
XA	rp'	—	XA < rp'	XA >= rp'	○	×
rp'	rp'	XA	rp' < rp'	rp'>= rp'	○	×
#n8	rp'	XA	n8 < rp'	n8 >= rp'	○	×
@HL	rp'	XA	(HL) < rp'	(HL) >= rp'	○	×
mem	rp'	XA	(mem) < rp'	(mem) >= rp'	○	×
rp'1	XA	—	rp'1 < XA	rp'1 >= XA	○	×
★ #n8	XA	rp'1	n8 < XA	n8 >= XA	○	×

<2> If a register name is not specified, the contents of α are rewritten as $\alpha - \beta$.

[Generated Instructions]

<1> When register is not specified ($\alpha<\beta$)

```
SUBS    α, β
BR      ?LFALSE
```

<2> When register is specified ($\alpha<\beta$ (γ))

```
MOV     γ, α
SUBS    γ, β
BR      ?LFALSE
```

Compare Operator

[Example]

<Input source program>

```
if ( A < @HL )
            CALL    !XXX
else
            CALL    !YYY
endif
if ( B < @HL ) ( A )
            CALL    !PPP
endif
```

<Output source program>

```
                SUBS    A,@HL   ;if ( A < @HL )
                BR      ?L1
                CALL    !XXX    ;           CALL    !XXX
                BR      ?L2
?L1:                            ;else
                CALL    !YYY    ;           CALL    !YYY
?L2:                            ;endif
                MOV     A,B     ;if ( B < @HL ) ( A )
                SUBS    A,@HL
                BR      ?L3
                CALL    !PPP    ;           CALL    !PPP
?L3:                            ;endif
```

Compare Operator

(4) >

[Format]

$\alpha > \beta$ [$\triangle$ (γ)] γ: register name

[Purpose]

<1> True if the contents of α are greater than those of β; otherwise, false.

<2> If a register name is specified, the contents of β are transferred to γ. True if the contents of α are greater than those of the register; otherwise, false.

[Explanation]

<1> Acceptable formats for α, β, and γ and the cases where the result is true and false are shown below.

α	β	γ	True	False	H	S
@HL	A	—	(HL) > A	(HL) <= A	○	○
@HL	reg1	A	(HL) > reg1	(HL) <= reg1	○	○
@HL	#n4	A	(HL) > n4	(HL) <= n4	○	○
@HL	@rpa1	A	(HL) > (rpa1)	(HL) <= (rpa1)	○	○
@HL	mem	A	(HL) > (mem)	(HL) <= (mem)	○	○
rp'	XA	—	rp' > XA	rp' <= XA	○	×
rp'	rp'	XA	rp' > rp'	rp' <= rp'	○	×
rp'	#n8	XA	rp' > n8	rp' <= n8	○	×
rp'	@HL	XA	rp' > (HL)	rp' <= (HL)	○	×
rp'	mem	XA	rp' > (mem)	rp' <= (mem)	○	×
XA	rp'1	—	XA > rp'	XA <= rp'1	○	×
★ XA	#n8	rp'1	XA > n8	XA <= n8	○	×

<2> If a register name is not specified, the contents of β are rewritten as $\beta - \alpha$.

[Generated Instructions]

<1> When register is not specified ($\alpha > \beta$)

```
SUBS    β, α
BR      ?LFALSE
```

<2> When register is specified ($\alpha > \beta$ (γ))

```
MOV     γ, β
SUBS    γ, α
BR      ?LFALSE
```

Compare Operator

[Example]

<Input source program>

```
if ( BC > XA )
            CALL    !XXX
else
            CALL    !YYY
endif
if ( BC > DE ) ( XA )
            CALL    !PPP
endif
```

<Output source program>

```
                SUBS    XA,BC   ;if ( BC > XA )
                BR      ?L1
                CALL    !XXX    ;           CALL    !XXX
                BR      ?L2
?L1:                            ;else
                CALL    !YYY    ;           CALL    !YYY
?L2:                            ;endif
                MOV     XA,DE   ;if ( BC > DE ) ( XA )
                SUBS    XA,BC
                BR      ?L3
                CALL    !PPP    ;           CALL    !PPP
?L3:                            ;endif
```

Compare Operator

(5) >=

[Format]

α >= β [$\triangle$ (γ)] γ: register name

[Purpose]

<1> True if the contents of α are equal to or greater than those of β; otherwise, false.

<2> If a register name is specified, the contents of α are transferred to the specified register. True if the contents of the specified register are equal to or greater than those of β; otherwise, false.

[Explanation]

<1> Acceptable formats for α, β, and γ and the cases where the result is true and false are shown below.

α	β	γ	True	False	H	S
A	@HL	—	A >= (HL)	A < (HL)	○	○
reg1	@HL	A	reg1 >= (HL)	reg1 < (HL)	○	○
#n4	@HL	A	n4 >= (HL)	n4 < (HL)	○	○
@rpa1	@HL	A	(rpa1) >= (HL)	(rpa1) < (HL)	○	○
mem	@HL	A	(mem) >= (HL)	(mem) < (HL)	○	○
XA	rp'	—	XA >= rp'	XA < rp'	○	×
rp'	rp'	XA	rp' >= rp'	rp' < rp'	○	×
#n8	rp'	XA	n8 >= rp'	n8 < rp'	○	×
@HL	rp'	XA	(HL) >= rp'	(HL) < rp'	○	×
mem	rp'	XA	(mem) >= rp'	(mem) < rp'	○	×
rp'1	XA	—	rp'1 >= XA	rp'1 < XA	○	×
★ #n8	XA	rp'1	n8 >= XA	n8 < XA	○	×

<2> If a register name is not specified, the contents of α are rewritten as $\alpha - \beta$.

[Generated Instructions]

<1> When register is not specified (α>=β)

```
SUBS    α, β
BR      ?LTRUE
BR      ?LFALSE
```

<2> When register is specified (α>=β (γ))

```
MOV     γ, α
SUBS    γ, β
BR      ?LTRUE
BR      ?LFALSE
```

Compare Operator

[Example]

<Input source program>

```
if ( BC >= DE ) ( XA )
            CALL    !XXX
else
            CALL    !YYY
endif
```

<Output source program>

```
                MOV     XA,BC    ;if ( BC >= DE ) ( XA )
                SUBS    XA,DE
                BR      ?L2
                BR      ?L1
?L2:
                CALL    !XXX     ;            CALL        !XXX
                BR      ?L3
?L1:                             ;else
                CALL    !YYY     ;            CALL        !YYY
?L3:                             ;endif .
```

Compare Operator

(6) <=

[Format]

α <= β [$\triangle$ (γ)] γ: register name

[Purpose]

<1> True if the contents of α are equal to or less than those of β; otherwise, false.

<2> If a register name is specified, the contents of β are transferred to the specified register. True if α is equal to or less than the contents of the specified register; otherwise, false.

[Explanation]

<1> Acceptable formats for α, β, and γ and the cases where the result is true and false are shown below.

α	β	γ	True	False	H	S
@HL	A	—	(HL) <= A	(HL) > 1	○	○
@HL	reg1	A	(HL) <= reg1	(HL) > reg1	○	○
@HL	#n4	A	(HL) <= n4	(HL) > n4	○	○
@HL	@rpa1	A	(HL) <= (rpa1)	(HL) > (rpa1)	○	○
@HL	mem	A	(HL) <= (mem)	(HL) > (mem)	○	○
rp'	XA	—	rp' <= XA	rp' > XA	○	×
rp'	rp'	XA	rp' <= rp'	rp' > rp'	○	×
rp'	#n8	XA	rp' <= n8	rp' > n8	○	×
rp'	@HL	XA	rp' <= (HL)	rp' > (HL)	○	×
rp'	mem	XA	rp' <= (mem)	rp' > (mem)	○	×
XA	rp'1	—	XA <= rp'1	XA > rp'1	○	×
★ XA	#n8	rp'1	XA <= n8	XA > n8	○	×

<2> If a register name is not specified, the contents of β are rewritten as $\beta - \alpha$.

[Generated Instructions]

<1> When register is not specified (α<=β)

```
SUBS    β, α
BR      ?LTRUE
BR      ?LFALSE
```

<2> When register is specified (α<=β (γ))

```
MOV     γ, β
SUBS    γ, a
BR      ?LTRUE
BR      ?LFALSE
```

Compare Operator

[Example]

<Input source program>

```
if ( BC <= XA )
            CALL    !XXX
else
            CALL    !YYY
endif
```

<Output source program>

```
                SUBS    XA,BC   ;if ( BC <= XA )
                BR      ?L2
                BR      ?L1
?L2:
                CALL    !XXX    ;          CALL    !XXX
                BR      ?L3
?L1:                            ;else
                CALL    !YYY    ;          CALL    !YYY
?L3:                            ;endif
```

2.4.5 Logical operator

The result of a logical operator is true or false according to whether the first and second expressions are true or false.

A logical operator, and the first and second expressions constitute a logical operation expression.

Table 2-8 Types of Logical Operator lists the logical operators available.

Table 2-8. Types of Logical Operator

Operator	Format	Purpose
Logical product (&&)	α && β	True if α and β are true; otherwise, false.
Logical sum (||)	α || β	True if either or both of α or β are true; otherwise false.

Logical Operator

(1) Logical product (&&)

[Format]

Conditional expression 1 && conditional expression 2

[Purpose]

ANDs conditional expressions 1 and 2. True if both the expressions are true; otherwise, false.

[Explanation]

<1> A compare operation expression or bit symbol can be used for conditional expressions 1 and 2.

<2> If a register is specified in a control statement, the register is used for both conditional expressions 1 and 2. Therefore, different registers cannot be specified for expressions 1 and 2.

[Generated Instructions]

If conditional expression 1 is false, the result of the AND is false regardless of whether conditional expression 2 is true or false. Therefore, instructions that evaluate conditional expression 2 are generated only when conditional expression 1 is true.

For details of the generated instructions, refer to **2.4.7 Label generation rules.**

[Example]

<Input source program>

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

<Output source program>

```
                SKE     A,#0    ;if ( A == #0 && B != #0 )
                BR      ?L1
                SKE     B,#0
                BR      ?L2
                BR      ?L1
?L2:
                CALL    !XXX    ;          CALL    !XXX
                BR      ?L3
?L1:                            ;else
                CALL    !YYY    ;          CALL    !YYY
?L3:                            ;endif
```

Logical Operator

(2) Logical sum (||)

[Format]

Conditional expression 1 || conditional expression 2

[Purpose]

ORs conditional expressions 1 and 2. True if one or both of the expressions are true; otherwise, false.

[Explanation]

<1> A compare operation expression or bit symbol can be used for conditional expressions 1 and 2.

<2> If a register is specified in a control statement, the register is used for both conditional expressions 1 and 2. Therefore, different registers cannot be specified for expressions 1 and 2.

[Generated Instruction]

If conditional expression 1 is true, the result of the OR is true regardless of whether conditional expression 2 is true or false. Therefore, instructions that evaluate conditional expression 2 are generated only when conditional expression 1 is false.

For details of the generated instruction, refer to **2.4.7 Label generation rules.**

[Example]

<Input source program>

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

<Output source program>

```
                SKE     A,#0    ; if ( A == #0 || C != #0 )
                BR      ?L1
                BR      ?L2
?L1:
                SKE     C,#0
                BR      ?L2
                BR      ?L3
?L2:
                CALL    !XXX    ;               CALL    !XXX
                BR      ?L4
?L3:                            ;else
                CALL    !YYY    ;               CALL    !YYY
?L4:                            ;endif
```

2.4.6 Bit condition

A bit address written in a conditional statement used for bit test is called a bit condition.

A bit condition may be either of the following two types:

- Bit address
- ! bit address

A bit condition must not be used alone. Be sure to use it in a control statement.

★ **Remark** The structured assembler recognizes a user symbol as a bit address. It does not check whether a bit address has been defined by the symbol pseudoinstruction (EQU instruction) of the assembly language.

Bit Address

(1) Bit address

[Format]

```
if_bit (bit address)
while_bit (bit address)
until_bit (bit address)
```

[Purpose]

True if the contents of the bit address are "H"; false if they are "L".

[Explanation]

Acceptable bit addresses and the cases where the result is true and false are shown below.

Bit Address	True	False	H	S
mem.bit	(mem.bit) = 1	(mem.bit) = 0	○	○
fmem.bit	(fmem.bit) = 1	(fmem.bit) = 0	○	○
pmem.@L	(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$)) = 1	(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$)) = 0	○	○
@H+mem.bit	(H+mem.bit) = 1	(H+mem.bit) = 0	○	Note
CY	CY = 1	CY = 0	○	○

Note This cannot be used with the μPD75048 when MSB = 4, 5, 6, or 7.

[Generated Instructions]

```
SKT     Bit address
BR      ?LFALSE
```

[Example]

<Input source program>

```
if_bit ( TRFG.0 )
    CALL    !XXX
else
    CALL    !YYY
endif
```

<Output source program>

```
                SKT     TRFG.0  ;if_bit ( TRFG.0 )
                BR      ?L1
                CALL    !XXX    ;   CALL    !XXX
                BR      ?L2
?L1:                            ;else
                CALL    !YYY    ;   CALL    !YYY
?L2:                            ;endif
```

!Bit Address

(2) ! bit address

[Format]

```
if_bit (! bit address)
while_bit (! bit address)
until_bit (! bit address)
```

[Purpose]

True if the contents of the bit address are "L"; false if they are "H".

[Explanation]

Acceptable bit addresses and the cases where the result is true and false are shown below.

Bit Address	True	False	H	S
mem.bit	(mem.bit) = 0	(mem.bit) = 1	○	○
fmem.bit	(fmem.bit) = 0	(fmem.bit) = 1	○	○
pmem.@L	(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$)) = 0	(pmem$_{7-2}$+L$_{3-2}$.bit (L$_{1-0}$)) = 1	○	○
@H+mem.bit	(H+mem.bit) = 0	(H+mem.bit) = 1	○	Note
CY	CY = 0	CY = 1	○	○

Note This cannot be used with the μPD75048 when MSB = 4, 5, 6, or 7.

[Generated Instructions]

<1> If CY is used as a bit address

```
SKT     CY
BR      ?LTRUE
BR      ?LFALSE
```

<2> Other than <1>

```
SKF     Bit address
BR      ?LFALSE
```

!Bit Address

[Example]

<Input source program>

```
if_bit ( !TRFG.0 )
    CALL    !XXX
else
    CALL    !YYY
endif
```

<Output source program>

```
                SKF     TRFG.0  ;if_bit ( !TRFG.0 )
                BR      ?L1
                CALL    !XXX    ;       CALL    !XXX
                BR      ?L2
?L1:                            ;else
                CALL    !YYY    ;       CALL    !YYY
?L2:                            ;endif
```

2.4.7 Label generation rules

As a control statement is processed, a label for a branch instruction is generated.
The rules according to which the label is generated are explained below.

(1) Generated label is "?Ldddd".

dddd is a decimal number starting from 1 and is output, zero-suppressed and left-justified. Therefore, do not use labels that start with "?L".

(2) Label generation rule for logical operation expressions

Label generation for logical sum (||) and logical product (&&) is classified into the following patterns depending on the type of the two members, α and β.
In the following explanation,
If the conditional expression is ==, <, >, or bit symbol (except !CY) : C1
If the conditional expression is !=, <=, >=, or !CY : C2
The jump destination label when the control statement is true is ?Ltrue, and that when the control statement is false is ?Lfalse.

<1> If (C1 && C1)

```
        Conditional expression test instruction of C1
        BR      ?Lfalse
        Conditional expression test instruction of C2
        BR      ?Lfalse
```

<2> If (C1 && C2)

```
        Conditional expression test instruction of C1
        BR      ?Lfalse
        Conditional expression test instruction of C2
        BR      ?Ltrue
        BR      ?Lfalse
?Ltrue:
```

<3> If (C2 && C1)

```
        Conditional expression test instruction of C2
        BR      ?Ltrue
        BR      ?Lfalse
?Ltrue
        Conditional expression test instruction of C1
        BR      ?Lfalse
```

<4> If (C2 && C2)

```
        Conditional expression test instruction of C2
        BR      ?Ltrue1
        BR      ?Lfalse
?Ltrue1:
        Conditional expression test instruction of C2
        BR      ?Ltrue2
        BR      ?Lfalse
```

<5> If (C1 || C1)

```
        Conditional expression test instruction of C1
        BR      ?Lfalse1
        BR      ?Ltrue
?Lfalse1:
        Conditional expression test instruction of C1
        BR      ?Lfalse2
?Ltrue:
```

<6> If (C1 || C2)

```
        Conditional expression test instruction of C1
        BR      ?Lfalse1
        BR      ?Ltrue
?Lfalse1:
        Conditional expression test instruction of C2
        BR      ?Ltrue
        BR      ?Lfalse2
?Ltrue:
```

<7> If (C2 || C1)

```
        Conditional expression test instruction of C2
        BR      ?Ltrue
        Conditional expression test instruction of C1
        BR      ?Lfalse
?Ltrue:
```

<8> If (C2 || C2)

```
        Conditional expression test instruction of C2
        BR      ?Ltrue
        Conditional expression test instruction of C2
        BR      ?Ltrue
        BR      ?Lfalse
?Ltrue:
```

3.1 Outline of Control Statements

Control statements are used so that the flow of program control can be written in a structured way. The following control statements are used.

(1) IF ~ THEN ~ ELSE

Two-branch structure	if~else~endif if_bit~else~endif
Multi-branch structure	if~elseif~else~endif[Note] if_bit~elseif_bit~else~endif[Note] switch~case~default~ends

Note Two or more elseif and elseif_bit can be used.

(2) DO ~ WHILE

Repetition of increment statement	for~next
Repetition of preprocessing condition test	while~endw while_bit~endw
Repetition of postprocessing condition	repeat~until repeat~until_bit
Repetition of and exit from loop block	continue break
Escape for exception processing	goto
★ If loop statement is indefinite loop	forever

3.2 Nesting

Control statements can be nested. The nesting level is up to 31 statements. Take care, however, that control statements do not cross.

(Incorrect coding)

```
while (XA<BC)
  if (A==#4)
        break;
endw
      XA++
  endif
```

Because control statements are crossed, an error occurs.

(Correct coding)

```
while (XA<BC)
  if (A==#4)
      break;
  endif
  XA++
endw
```

IF statement is correctly nested in WHILE statement.

★ 3.3 Register Specification

This section explains register specification.

Register Name Specification

[Format]

(register name)

[Purpose]

The meaning differs depending on at which position of a control statement a register is specified.

<1> If a register is specified immediately after the conditional expression of a compare operator.
An instruction that transfers the left member to the specified register is generated and then an instruction that compares the specified register with the right member is generated.

Example

```
            SKE     B,#5        ;if ( B != #5 && mem >= @HL (A) )
            BR      ?L1
            BR      ?L2
?L1:
            MOV     A,mem
            SUBS    A,@HL
            BR      ?L3
            BR      ?L2
?L3:
            CALL    !XXX        ;       CALL    !XXX
            BR      ?L4
?L2:                            ;else
            CALL    !YYY        ;       CALL    !YYY
?L4:                            ;endif
```

<2> If a register is specified after a control statement
An instruction that transfers the left term to the specified register is generated when the instruction of the conditional expression of each compare operator is generated, and then an instruction that compares the specified register with the right term is generated.

Example

```
            MOV     A,#4        ;if ( #4 != #5 && mem >= @HL ) (A)
            SKE     A,#5
            BR      ?L1
            BR      ?L2
?L1:
            MOV     A,mem
            SUBS    A,@HL
            BR      ?L3
            BR      ?L2
?L3:
            CALL    !XXX        ;       CALL    !XXX
            BR      ?L4
?L2:                            ;else
            CALL    !YYY        ;       CALL    !YYY
?L4:                            ;endif
```

Register Name Specification

<3> If both <1> and <2> apply

The register specified immediately after the conditional expression of each compare operator takes precedence. An instruction that transfers the left term to the specified register is generated, and then an instruction that compares the specified register with the right term is generated.

Example

```
            MOV     X,#4        ;if (#4 != #5 && mem >= @HL (A)) (X)
            SKE     X,#5
            BR      ?L1
            BR      ?L2
?L1:
            MOV     A,mem
            SUBS    A,@HL
            BR      ?L3
            BR      ?L2
?L3:
            CALL    !XXX        ;          CALL    !XXX
            BR      ?L4
?L2:                            ;else
            CALL    !YYY        ;          CALL    !YYY
?L4:                            ;endif
```

[Explanation]

This can be done with the if, elseif, switch, for, while, and until statements. However, if the conditional statement is a bit conditional statement, and if increment or decrement is used in the assignment statement in the for statement, the register specified by the control statement is ignored.

3.4 Purpose of Control Statements

The purpose of control statements is explained below.

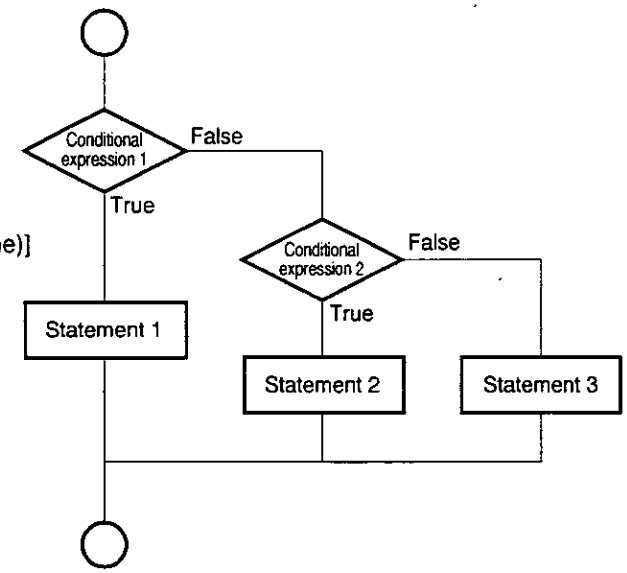
Note that two or more statements can be included in the portion indicated as "statement" in the following explanation.

if~elseif~else~endif

(1) if ~ elseif ~ else ~ endif

[Format]

```
if (conditional expression 1) [(register name)]
        Statement 1
elseif (conditional expression 2) [(register name)]
        Statement 2
else
        Statement 3
endif
```



[Purpose]

<1> if~endif
Executes statement 1 if conditional expression 1 is true.

<2> if~else~endif
Executes statement 1 if conditional expression 1 is true; if it is false, executes statement 3.

<3> if~elseif~else~endif
Executes statement 1 if conditional expression 1 is true; if it is false, tests conditional statement 2. If conditional statement 2 is true, executes statement 2; if it is false, executes statement 3.

[Explanation]

<1> if~else~endif is used to choose one of two branches depending on a given condition.

<2> if~elseif~else~endif is used to choose one of a number of possible branches depending on a range of values. These statements differ from the switch statement in that they can have a range of values.

<3> A compare operation expression or a logical operation expression can be used as a conditional expression. If a register name is specified, the specified register is used for the condition test.

<4> The elseif and else statements may be omitted. Two or more elseif statement may be used.

if~elseif~else~endif

[Generated Instructions]

<1> Processing of if (conditional expression)

Generates an instruction that tests the condition of a conditional expression.

<2> Processing of elseif (conditional expression)

(a) Generates an instruction that branches to the endif statement.

(b) Generates a label for the branch instruction that is generated by the if statement.

(c) Generates an instruction that tests the condition of a conditional expression.

<3> Processing of else

(a) Generates an instruction that branches to the endif statement.

(b) Generates a label for the branch instruction that is generated by the if and elseif statements.

<4> Processing of endif

Generates a label for the branch instruction generated by the if, elseif, and else statements.

if~elseif~else~endif

[Example]

<Input source program>

```
if ( B == #0 )
       TMOD0 = XA
       XA = #0CH
else
       XA = #0AH
endif
       TM0 = XA
```

<Output source program>

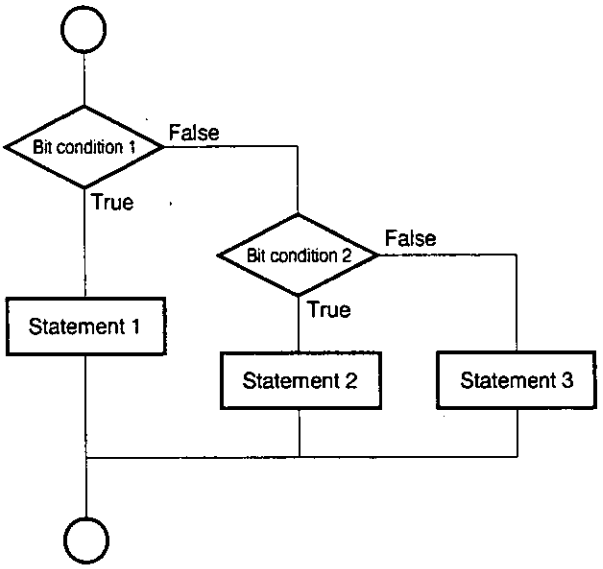
```
          SKE      B, #0          ;if ( B == #0 )
          BR       ?L1
          MOV      TMOD0,XA       ;       TMOD0 = XA
          MOV      XA,#0CH        ;       XA = #0CH
          BR       ?L2
?L1:                              ;else
          MOV      XA,#0AH        ;       XA = #0AH
?L2:                              ;endif
          MOV      TM0,XA         ;       TM0 = XA
```

if_bit~elseif_bit~else~endif

(2) if_bit~elseif_bit~else~endif

[Format]

```
if_bit (bit conditional expression 1)
        Statement 1
elseif_bit (bit conditional expression 2)
        Statement 2
else
        Statement 3
endif
```



[Purpose]

<1> if_bit~endif
Executes statement 1 if bit condition 1 is true.

<2> if_bit~else~endif
Executes statement 1 if bit condition 1 is true; if it is false, executes statement 3.

<3> if_bit~elseif_bit~else~endif
Executes statement 1 if bit condition 1 is true; if it is false, tests bit condition 2. Executes statement 2 if bit condition 2 is true; if it is false, executes statement 3.

[Explanation]

<1> if_bit~else~endif is used to choose one of two branches depending on a given condition.

<2> if_bit~elseif_bit~else~endif is used to check two or more bit symbols and to execute one of a number of branches.

<3> The elseif_bit and else statements may be omitted. Two or more elseif_bit statements may be used.

if_bit~elseif_bit~else~endif

[Generated Instructions]

<1> Processing of if_bit (bit condition)

Generates an instruction that tests whether a bit condition is true or false.

<2> Processing of elseif_bit (bit condition)

(a) Generates an instruction that branches to the endif statement.

(b) Generates a label for the branch instruction that is generated by the if_bit statement.

(c) Generates an instruction that tests whether a bit condition is true or false.

<3> Processing of else

(a) Generates an instruction that branches to the endif statement.

(b) Generates a label for the branch instruction that is generated by the if_bit and elseif_bit statements.

<4> Processing of endif

Generates a label for the branch instruction generated by the if_bit, elseif_bit, and else statements.

[Example]

<Input source program>

```
INTSIO:
if_bit ( !TRFG.O )
     CNT = #0EH ( A )
     SET1     PRTYFLG.3
elseif_bit ( PGF.0 )
     SIO = RDATA ( XA )
     XA = #0FFH
else
     H = #(FG SHR 6)
     CY = @H+PFG.0
     CLR1     BUSYFG.2
endif
```

if_bit~elseif_bit~else~endif

<Output source program>

```
INTSIO:                              ;INTSIO:
        SKF     TRFG.0               ;if_bit ( !TRFG.0 )
        BR      ?L1
        MOV     A,#0EH               ;   CNT = #0EH ( A )
        MOV     CNT, A
        SET1    PRTYFLG.3            ;   SET1    PRTYFLG.3
        BR      ?L2
?L1:                                 ;elseif_bit ( PGF.0 )
        SKT     PGF.0
        BR      ?L3
        MOV     XA,RDATA             ;   SIO = RDATA ( XA )
        MOV     SIO,XA
        MOV     XA,#0FFH             ;   XA = #0FFH
        BR      ?L2
?L3:                                 ;else
        MOV     H,#(FG SHR 6)        ;   H = #(FG SHR 6)
        MOV1    CY,@H+PFG.0          ;   CY = @H+PFG.0
        CLR1    BUSYFG.2             ;   CLR1    BUSYFG.2
?L2:                                 ;endif
```

switch~case~default~ends

(3) switch~case~default~ends

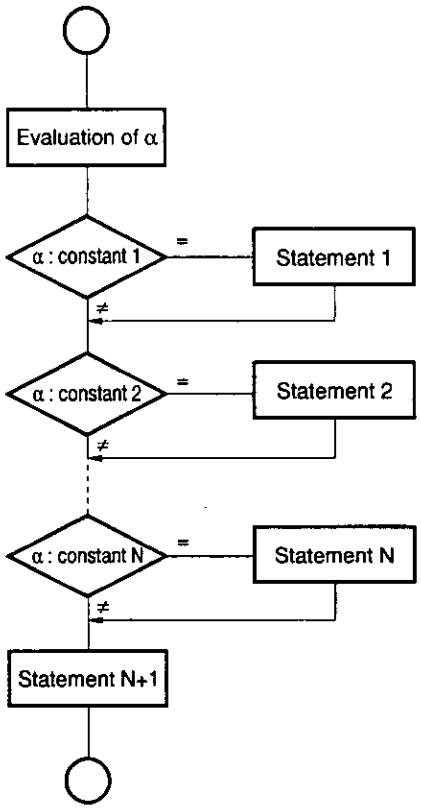
[Format]

★

```
switch (α) [(γ)]
    case  constant 1:
                    Statement 1
    case  constant 2:
                    Statement 2
          :
    case  constant N:
                    Statement N

    default:
                    Statement N+1
ends
```

Remark α: Symbol
γ: Register name



[Purpose]

Executes statement i when the value of α coincides with constant i (i = 1 to N). If the value of α does not coincide with constant 1 to N, and if a default is used, executes statement N+1. If the value of α does not coincide with constant 1 to N, and if a default is not used, nothing is executed.
Usually, a break statement must be included to exit from the switch block.

switch~case~default~ends

[Explanation]

<1> The following can be used as α.

α	γ	Operation	H	S
#n4	—	if #n4 = constant i then goto statement i	○	○
@HL[Note]	—	if @HL = constant i then goto statement i	○	○
@rpa1	—	if @rpa1 = constant i then goto statement i	○	○
mem	—	if mem = constant i then goto statement i	○	○
reg[Note]	—	if reg = constant i then goto statement i	○	○
A	reg	if A = constant i then goto statement i	○	○
A	@HL	if A = constant i then goto statement i	○	○
reg1	A	if reg1 = constant i then goto statement i	○	○
@HL	A	if @HL = constant i then goto statement i	○	○
@rpa1	A	if @rpa1 = constant i then goto statement i	○	○
#n4	reg	if #n4 = constant i then goto statement i	○	○
mem	A	if mem = constant i then goto statement i	○	○

Note The transfer instruction (MOV instruction) is not generated. (i = 1 to N)

<2> After control has been transferred to the case statement, the subsequent case statements are sequentially executed. Therefore, include a break statement to exit from the switch statement without transferring control to the next case statement.

<3> As a constant, a binary, octal, decimal, hexadecimal, or character constant can be used. Because the structured assembler recognizes a constant as a character string, however, the constant must be one that can be interpreted by the assembler as a constant.

<4> Include a default statement at the end of case. If it is written before case, an error message is output.

<5> If no register is specified, the contents of the value of "A (accumulator)" are changed. To retain the value of "A", save the value of "A" before the switch statement.

[Generated Instructions]

<1> Processing of switch

(a) If no register is specified (switch (α))

```
MOV     A, α
```

(b) If a register is specified (switch (α) (γ))

```
MOV     γ, α
```

<2> Processing of case

(a) Generates a label for the branch instruction that is generated by the preceding case statement.

(b) Generates the following instructions:

```
SKE     A (or γ), #constant i
BR      ?Lfalse
```

<3> Processing of default

Generates a label for the branch instruction that is generated by the case statement.

<4> Processing of ends

Generates a label for the branch instructions that are generated by the case and break statements.

switch~case~default~ends

[Example]

<Input source program>

```
switch ( MODE1 )
    case 1:
        if_bit ( PORT1.0 )
            SET1    BTM.3
        endif
        break
    case 2:
        BRCB    !LEADA
        break
    case 3:
        BRCB    !DATACD
        break
    default:
        BRCB    !REPTA
ends
```

<Output source program>

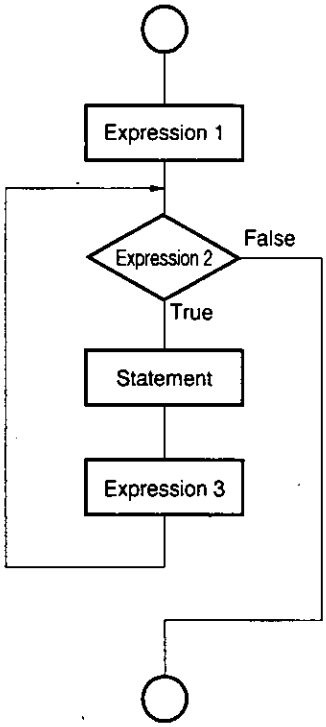
```
        MOV     A,MODE1  ;switch ( MODE1 )
        SKE     A,#1     ;  case 1:
        BR      ?L1
        SKT     PORT1.0  ;       if_bit ( PORT1.0 )
        BR      ?L2
        SET1    BTM.3    ;           SET1     BTM.3
?L2:                     ;       endif
        BR      ?L3      ;       break
?L1:                     ;  case 2:
        SKE     A,#2
        BR      ?L4
        BRCB    !LEADA   ;       BRCB     !LEADA
        BR      ?L3      ;       break
?L4:                     ;  case 3:
        SKE     A,#3
        BR      ?L5
        BRCB    !DATACD  ;       BRCB     !DATACD
        BR      ?L3      ;       break
?L5:                     ;  default:
        BRCB    !REPTA   ;       BRCB     !REPTA
?L3:                     ;ends
```

for~next

(4) for ~ next

[Format]

```
for (expression 1: expression 2: expression 3) [(register name)]
        Statement
next
```



[Purpose]

Sets an initial value with expression 1, and executes the statement and expression 3 while the conditional expression of expression 2 is satisfied.

[Explanation]

<1> Define an initial value for expression 1 (assignment expression), a conditional expression for expression 2, and an assignment expression such as increment/decrement for expression 3.

★ <2> Execution enters an indefinite loop if "forever" is used in the conditional expression.

<3> A compare operation expression or logical operation expression can be used as a conditional expression.

<4> If a register name is specified, the register is used to assign expression 1 and test the conditional expression of expression 2. Therefore, only "A" and "XA" can be used as the register name.

★ <5> Expressions 1, 2, and 3 may be omitted. If expression 2 is omitted, execution enters an indefinite loop.

<6> Expressions 2 and 3 control this block, and their contents must not be changed by an execution statement. If they are changed, malfunctioning may occur.

<7> The meaning is equivalent to the following. However, expansion differs depending on the instructions generated.

```
Expression 1
  while (expression 2)
        Statement
        Expression 3
  endw
```

for~next

[Generated Instructions]

<1> Processing of for (expression 1; expression 2; expression 3)

(a) Generates the instruction of expression 1.

(b) Generates an instruction that branches to the statement that tests the condition of expression 2.

(c) Generates a label for the branch instruction generated by the next statement.

(d) Generates an instruction for the assignment expression of expression 3.

(e) Generates a label for the branch instruction generated in (b).

(f) Generates the condition test instruction of expression 2.

<2> Processing of next

(a) Generates a branch instruction for the label generated in (c) of processing of the for statement.

(b) Generates a label for the branch instruction that is used to exit from the for block.

[Example]

<Input source program>

```
BC = #80H
for ( i = #0H; i < BC ; i++ ) ( XA )
    CALL!XXX
next
```

<Output source program>

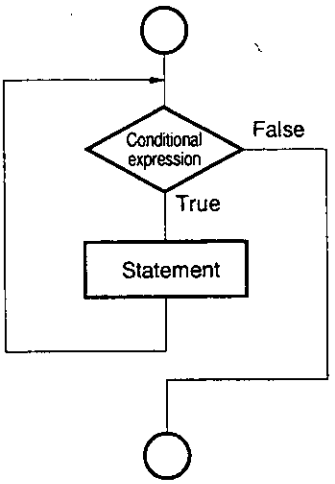
```
        MOV     BC,#80H  ;BC = #80H
        MOV     XA,#0H   ;for ( i = #0H ; i < BC ; i++ ) ( XA )
        MOV     i,XA
?L1:
        MOV     XA,i
        SUBS    XA,BC
        BR      ?L2
        CALL    !XXX     ;    CALL     !XXX
        INCS    i
        BR      ?L1
?L2:                     ;next
```

while~endw

(5) while~endw

[Format]

```
while (conditional expression) [(register name)]
      Statement
endw
```



[Purpose]

Executes the statement while the conditional expression is true.

[Explanation]

<1> The statement is never executed if the conditional expression is initially false because the conditional expression is evaluated before the statement is executed.

<2> Execution enters an indefinite loop if "forever" is used in the conditional expression.

<3> A compare operation expression or logical operation expression can be used as the conditional expression. If a register name is specified, the condition is tested using the register.

[Generated Instructions]

<1> while (conditional expression)

(a) Generates a label for the branch instruction that is generated by endw.

(b) Generates an instruction that tests the condition of the conditional expression. If a register name is specified, the register is used to generate the instruction that tests the condition.

<2> endw

(a) Generates a branch instruction for repetition.

(b) Generates a label for the branch instruction that is used to exit from the while block.

while~endw

[Example]

<Input source program>

```
XA = #0H
while ( forever )
    A = #0
    OUT PORT4,A
    @HL = A
    if ( @HL == #0 )
        break
    endif
    A++
    @HL = A
    HL++
endw
```

<Output source program>

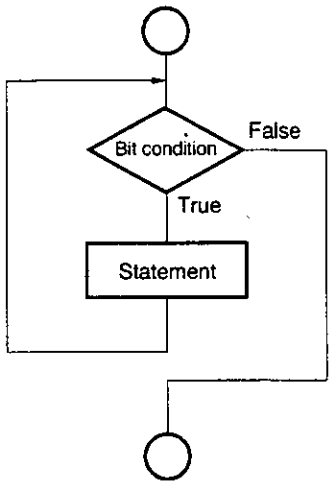
```
        MOV     XA,#0H   ;XA = #0H
?L1:                     ;while ( forever )
        MOV     A,#0     ;    A = #0
        OUT     PORT4,A  ;    OUT PORT4,A
        MOV     @HL,A    ;    @HL = A
        SKE     @HL,#0   ;    if ( @HL == #0 )
        BR      ?L2
        BR      ?L3      ;        break
?L2:                     ;    endif
        INCS    A        ;    A++
        MOV     @HL,A    ;    @HL = A
        INCS    HL       ;    HL++
        BR      ?L1
?L3:                     ;endw
```

while_bit~endw

(6) while_bit~endw

[Format]

```
while_bit (bit condition)
      Statement
endw
```



[Purpose]

Executes the statement while the bit condition is true.

[Description]

Because the bit condition is evaluated before the statement is executed, the statement is never executed if the bit condition expression is initially false.

[Generated Instruction]

<1> Processing of while_bit (bit condition)

(a) Generates a label for the branch instruction generated by endw.

(b) Generates an instruction that tests whether the bit condition is true or false.

<2> Processing of endw

(a) Generates a branch instruction for repetition.

(b) Generates a label for the branch instruction that is used to exit from the while_bit block.

while_bit~endw

[Example]

<Input source program>

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

<Output source program>

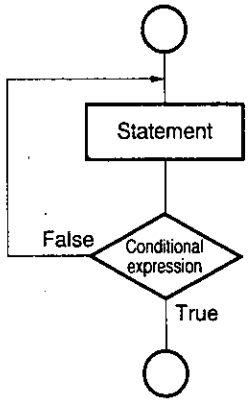
```
?L1:                            ;while_bit ( !TRFG.0 )
        SKF     TRFG.0
        BR      ?L2
        IN      A,PORT1         ;   IN      A,PORT1
        SKE     A,#4H           ;   if ( A == #4H )
        BR      ?L3
        MOV     XA, #0FFH       ;       XA = #0FFH
        BR      ?L4
?L3:                            ;   else
        CLR1    PFG.0           ;       CLR1    PFG.0
?L4:                            ;   endif
        BR      ?L1
?L2:                            ;endw
```

repeat~until

(7) repeat~until

[Format]

```
repeat
      Statement
until (conditional expression) [(register name)]
```



[Purpose]

Repeatedly executes the statement while the conditional expression is false.
The statement is executed at least once.

[Description]

<1> Evaluates the expression after the statement has been executed once.

<2> Execution enters an indefinite loop if "forever" is used in the conditional expression.

<3> A compare operation expression or logical operation expression can be used as the conditional expression. If a register name is specified, the specified register is used for condition testing.

[Generated Instructions]

<1> repeat
Generates a label for the branch instruction generated by until.

<2> until (conditional expression)
Generates an instruction that tests the condition of the conditional expression.

repeat~until

[Example]

<Input source program>

```
repeat
    @HL = XA
    if ( @HL != #0CH )
            CALL    !XXX
    endif
    HL++
until ( i == #0FH ) ( A )
```

<Output source program>

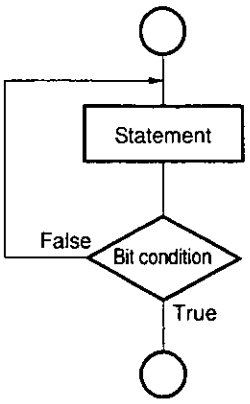
```
?L1:                            ;repeat
        MOV     @HL,XA          ;   @HL = XA
        SKE     @HL,#0CH        ;   if ( @HL != #0CH )
        BR      ?L3
        BR      ?L2
?L3:
        CALL    !XXX            ;       CALL    !XXX
?L2:                            ;   endif
        INCS    HL              ;   HL++
        MOV     A,i             ;until ( i == #0FH ) ( A )
        SKE     A,#0FH
        BR      ?L1
```

repeat~until_bit

(8) repeat~until_bit

[Format]

```
repeat
      Statement
until_bit (bit condition)
```



[Purpose]

Repeatedly executes the statement while the condition of the bit is false.
The statement is executed at least once.

[Explanation]

Evaluates the bit condition after the statement has been executed once.

[Generated Instructions]

<1> repeat
Generates a label for the branch instruction that is generated by until_bit.

<2> until_bit (bit conditional expression)
Generates an instruction that tests whether the bit conditional expression is true or false.

repeat~until_bit

[Example]

<Input source program>

```
repeat
    A = #8H
    OUT     PORT2,A
    CALL    !XXX
    IN      A,PORT0
until_bit ( TRIGF.0 )
```

<Output source program>

```
?L1:                        ;repeat
        MOV     A,#8H       ;   A = #8H
        OUT     PORT2,A     ;   OUT     PORT2,A
        CALL    !XXX        ;   CALL    !XXX
        IN      A, PORT0    ;   IN      A, PORT0
        SKT     TRIGF.0     ;until_bit ( TRIGF.0 )
        BR      ?L1
```

break

(9) break

[Format]

```
break
```

[Purpose]

Terminates execution of the innermost enclosed while, repeat, for, or switch block.

[Explanation]

If break is used in a statement other than while, while_bit, repeat~until, repeat~until_bit, for, or switch, the following error message is output to the secondary source file after translation:

```
*** illegal break statement ***
```

[Generated Instructions]

Generates a branch instruction that is used to exit from the while, repeat, for, or switch block.

★ BR ?LXXXX

[Example]

<Input source program>

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

<Output source program>

```
        MOV     XA,#0    ;XA = #0
?L1:                     ;while ( forever )
        MOV     A,#0     ;    A = #0
        OUT     PORT4,A  ;    OUT      PORT4,A
        SKE     A,#0FH   ;    if ( A == #0FH )
        BR      ?L2
        BR      ?L3      ;        break
?L2:                     ;    endif
        INCS    A        ;    A++
        BR      ?L1
?L3:                     ;endw
```

continue

(10) continue

[Format]

```
continue
```

[Purpose]

★ Skips the processing following continue in the innermost enclosed for, while, while_bit, until, or until_bit statement, and branches to the processing before condition testing.

[Explanation]

<1> Used to skip the subsequent processing in the middle of each block and repeatedly execute the next loop.

★ <2> If continue is used in a statement other than for, while, while_bit, until, or until_bit, the following error message is output to the secondary source file after translation:

```
*** illegal continue statement ***
```

★ **[Generated Instructions]**

Generates an instruction that branches to the label for loop repetition of the for, while, or until block.

```
BR      ?LXXXX
```

continue

[Example]

<Input source program>

```
XA = #0
while ( forever )
    A = #0
    OUT      PORT4,A
    CALL     !XXX
    if ( A == #8H )
        continue
    endif
    if ( C == #1 )
        break
    endif
    A++
endw
```

<Output source program>

```
        MOV     XA,#0    ;XA = #0
?L1:                     ;while ( forever )
        MOV     A,#0     ;    A = #0
        OUT     PORT4,A  ;    OUT      PORT4,A
        CALL    !XXX     ;    CALL     !XXX
        SKE     A,#8H    ;    if ( A == #8H )
        BR      ?L2
        BR      ?L1      ;        continue
?L2:                     ;    endif
        SKE     C,#1     ;    if ( C == #1 )
        BR      ?L3
        BR      ?L4      ;        break
?L3:                     ;    endif
        INCS    A        ;    A++
        BR      ?L1
?L4:                     ;endw
```

goto

(11) goto

[Format]

```
goto label
```

[Purpose]

Unconditionally branches to a label.

[Explanation]

<1> The goto statement is used if an error must be immediately processed, if it occurs, by an error-processing program, or if the processing is the same for errors that may occur at two or more locations.

<2> Specify the symbol used in the label field of the assembly language as the label.

[Generated Instructions]

The following instruction is generated:

```
BR      label
```

★ **Remark** The label of the goto statement is not automatically generated by the program. Therefore, the structured assembler does not output an error even if the label at the branch destination does not exist.

goto

[Example]

<Input source program>

```
XA = #0
while ( forever )
    A = #0
    OUT      PORT4,A
    CALL     !XXX
    if ( A == #0H )
        goto    ERROR
    endif
    A++
endw
```

<Output source program>

```
                MOV     XA,#0       ;XA = #0
?L1:                                ;while ( forever )
                MOV     A,#0        ;    A = #0
                OUT     PORT4,A     ;    OUT      PORT4,A
                CALL    !XXX        ;    CALL     !XXX
                SKE     A,#0H       ;    if ( A == #0H )
                BR      ?L2
                BR      ERROR       ;        goto     ERROR
?L2:                                ;    endif
                INCS    A           ;    A++
                BR      ?L1
                                    ;endw
```

forever

★ **(12) forever**

[Format]

- for (expression 1; forever; expression 3)
- while (forever)
- until (forever)

[Purpose]

Does not generate a compare instruction and uses the loop statement (for, while, or until) as an indefinite loop.

[Explanation]

The loop statement (for, while, or until) can be used in the conditional expression.

[Example]

<Input source program>

```
XA = #0
while ( forever )
    A = #0
    OUT     PORT4,A
    CALL    !XXX
    if ( A == #0H )
        goto    ERROR
    endif
    A++
endw
```

<Output source program>

```
            MOV     XA,#0       ;XA = #0
?L1:                            ;while ( forever )
            MOV     A,#0        ;    A = #0
            OUT     PORT4,A     ;    OUT      PORT4,A
            CALL    !XXX        ;    CALL     !XXX
            SKE     A,#0H       ;    if ( A == #0H )
            BR      ?L2
            BR      ERROR       ;        goto     ERROR
?L2:                            ;    endif
            INCS    A           ;    A++
            BR      ?L1
                                ;endw
```

CHAPTER 4 PSEUDOINSTRUCTIONS

4.1 Outline of Pseudoinstructions

Pseudoinstructions are written in the source program.
Pseudoinstructions give various directions necessary for the ST75X to perform certain processing.
By using pseudoinstructions, the source program can be more easily written.
Pseudoinstructions are not output to the output file.

4.2 Purpose of Pseudoinstructions

Table 4-1 List of Pseudoinstructions shows the types of pseudoinstructions available.

Table 4-1. List of Pseudoinstructions

Type of Pseudoinstruction	Pseudoinstruction
Symbol definition pseudoinstruction	#define
Conditional processing pseudoinstruction	#ifdef : #else : #endif
Include pseudoinstruction	#include
GETI replacement pseudoinstruction	#defgeti : #endgeti

The feature of each pseudoinstruction is explained below.

Symbol definition pseudoinstruction (#define)

(1) Symbol definition pseudoinstruction (#define)

[Format]

```
#define  symbol character  string
```

[Purpose]

Replaces a symbol used in the source program with a specified character string.

★ **[Explanation]**

<1> "#" character must be written first, apart from blanks and HT.

<2> A symbol must start with an alphabetic character and consist of alphanumeric characters. The valid number of characters for a symbol is 31 by default and 8 if the option "-NS" is specified. If a symbol of 9 characters or longer is specified when the valid number of characters is 8, the 9th character and those that follow are ignored. Likewise, if a symbol of 32 characters or longer is specified when the valid number of characters is 31, the 32nd character and those that follow are ignored.

<3> A character string can consist of the characters specified in 2.2.1 Character set. Blanks and quotation marks must not be used; if used, they are ignored and processing is continued.

<4> This pseudoinstruction is useful for writing a numeric value as an easy-to-read symbol.

<5> Reserved words must not be used as symbols.

<6> A reserved word can be used as a character string.

<7> If the same symbol is defined twice, a warning message is output.

<8> The translated character string is output to the secondary source file. The #define statement is not output.

<9> If the translated character string is defined by another #define, translation is performed again up to 31 times. If an attempt is made to execute translation 32 times or more, an error message is output, and the 32nd definition and those that follow are ignored.

<10> This pseudoinstruction can be used on any line in the source.

<11> If this pseudoinstruction is used with a symbol specified by the D option, a warning message is output, and #define takes precedence.

Symbol definition pseudoinstruction (#define)

[Example]

<Input source program>

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

<Output source program>

```
.................................................
                MOV     A,#0            ;A = #0
                SKE     A,#1            ;if ( A == #1 )
                BR      ?L1
                MOV     XA,#0C5H        ;       XA = #0C5H
?L1:                                    ;endif
```

Conditional processing pseudoinstruction (#ifdef/#else/#endif)

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

[Format]

```
#ifdef symbol
Text 1
#else
Text 2
#endif
```

[Purpose]

Executes conditional processing.

<1> If the value of the symbol is 0 or undefined
Skips text 1 and processes text 2.

<2> If the value of the symbol is other than 0
Processes text 1 and skips text 2.

[Explanation]

<1> "#" character must be written first, apart from blanks and HT.

<2> A symbol must start with an alphabetic character and consist of alphanumeric characters. The first 8 characters are valid.

<3> The symbol is defined in advance by the #define statement or by the D option if specified on starting.

<4> This pseudoinstruction can be nested up to 8 levels.

<5> #else may be omitted.

★ <6> This pseudoinstruction can be used on any line in the source program.

★ <7> This pseudoinstruction must not be used in a manner such that a control statement is divided or crossed.

[Example]

<Input source program (TEST.SRC))

```
#ifdef D75X
      Text 1
#else
      Text 2
#endif
```

<Starting method 1>

<Output source program (TEST.ASM))

```
A>ST75X TEST.SRC -DD75X   →   Text 2
```

In this case, text 2 is processed by the ST75X, and text 1 is skipped and not processed.

★ <Starting method 2>

<Output source program (TEST.ASM))

```
A>ST75X TEST.SRC   →   Text 1
```

In this case, text 1 is processed by ST75X and text 2 is skipped and not processed.

Include pseudoinstruction (#include)

(3) Include pseudoinstruction (#include)

[Format]

```
#include "file name"
```

[Purpose]

Replaces this one line with the contents of the specified file name and uses it as a source program to be processed by the ST75X.

[Explanation]

<1> "#" character must be written first apart from blanks and HT.

<2> This pseudoinstruction may be used on any line of the source program.

<3> The include pseudoinstruction cannot be used in the include file.
In other words, include must not be nested.

<4> The input source file name specified on starting, secondary source file names, and error file names must not be specified as a file name.

<5> A drive name and directory name can be written at the beginning of the file name. If these names are not given, it is assumed that the include file exists in the current drive and current directory.

<6> The drive name and directory of the include file can be specified by using the I option on starting the ST75X.

[Example]

<Input source program> <FILE>

```
#include "FILE"

        A=SIZE1
        B=SIZE2
```

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

<Output source program>

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

In this case, if -IB: \SRC\ is specified on starting, file in B: \SRC\FILE is read as #include "FILE".

GETI replacement pseudoinstruction (#defgeti)

(4) GETI replacement pseudoinstruction (#defgeti)

[Format]

```
#defgeti  label of GETI table
        instruction pattern
    :
#endgeti
```

[Purpose]

If an instruction pattern translated by the ST75X coincides with the specified instruction pattern, that instruction pattern is replaced by the GETI instruction for output to the secondary source file.

[Explanation]

<1> "#" character must be written first, apart from blanks and HT.

<2> This pseudoinstruction can be written up to 48 times.

<3> This pseudoinstruction is used to replace instructions translated by the ST75X.

<4> Because this pseudoinstruction matches patterns of instructions when the instructions are output to the secondary file, it can affect the processing performance if specified too many times. Therefore, avoid using this pseudoinstruction when it is not necessary.

<5> There are four possible instruction pattern combinations:

(a) Subroutine call instruction to anywhere in the internal ROM area

(b) Branch instruction to anywhere in the internal ROM area

(c) Any 2-byte, 2-machine cycle instruction (except the BRCB and CALLF instructions)

(d) Combination of two 1-byte instructions

However, the ST75X does not check whether the instructions can actually be combined.

<6> A pattern of two instructions takes precedence over a pattern of one instruction.

<7> The pattern defined first takes precedence if two patterns of the same two instructions or single instruction are defined. Therefore, even if the same pattern is defined twice, an error does not occur, and the pattern defined first becomes valid.

GETI replacement pseudoinstruction (#defgeti)

[Example]

<Input source program>

```
#defgeti   TXA_HL
           MOV      XA,@HL
#endgeti

C1         CSEG     IENT
TXA_HL:    MOV      XA,@HL

CN         CSEG
           XA = @HL
```

<Output source program>

```
C1                  CSEG    IENT    ;C1         CSEG    IENT

TXA_HL:             MOV     XA,@HL  ;TXA_HL:MOV         XA,@HL
CN                  CSEG            ;CN         CSEG
                    MOV     XA,@HL  ;           XA = @HL
```

[MEMO]

★

CHAPTER 5 CONTROL INSTRUCTIONS

5.1 Outline of Control Instructions

Control instructions give directions necessary for the ST75X to perform certain processing and are written in the source program.

By using control instructions, it becomes unnecessary to specify options when starting the program.

5.2 Assembler Control Instructions

It is checked whether an assembler control instruction can be written as the module header of the source module.

Table 5-1 lists the control instructions that can be specified only as a module header, and Table 5-2 shows the control instructions that are not recognized as a module header.

If a control instruction that can be specified only as a module header is used in a module body, an error occurs.

A control instruction that is not recognized as a module header can be written in both a module header and module body.

Remark A source module is defined as a module which is the input unit to the assembler when one source program is divided into several modules (if a program consists of only one module, source program and source module are the same in meaning).

This source module consists of the following three parts:

(1) Module header
(2) Module body
(3) Module tail

Figure 5-1. Configuration of Source Module

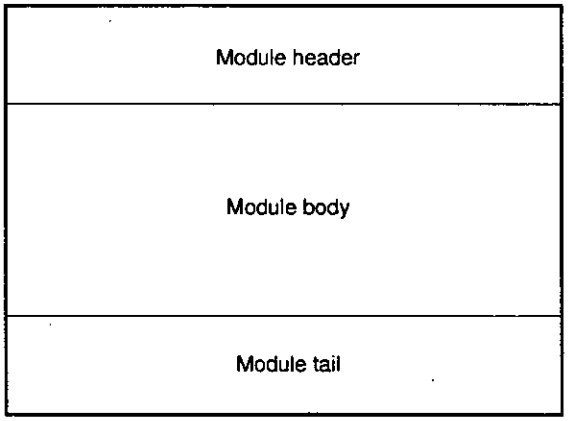


Table 5-1. Control Instructions That Can Be Specified Only for Module Header

Control Instruction
$ [△] PROCESSOR [△] = [△] product name [△] [; comment] $ [△] PC [△] = [△] product name [△] [; comment]
$ [△] MODE [△] = [△] constant [△] [; comment] $ [△] MD [△] = [△] constant [△] [; comment]
$ [△] DEBUGA [△] [; comment] $ [△] DA [△] [; comment]
$ [△] NODEBUGA [△] [; comment] $ [△] NODA [△] [; comment]
$ [△] DEBUG [△] [; comment] $ [△] DB [△] [; comment]
$ [△] NODEBUG [△] [; comment] $ [△] NODB [△] [; comment]
$ [△] SYMBOLS [△] [; comment] $ [△] SB [△] [; comment]
$ [△] NOSYMBOLS [△] [; comment] $ [△] NOSB [△] [; comment]
$ [△] XREF [△] [; comment] $ [△] XR [△] [; comment]
$ [△] NOXREF [△] [; comment] $ [△] NOXR [△] [; comment]
$ [△] PAGELENGTH [△] = [△] constant [△] [; comment] $ [△] PL = [△] constant [△] [; comment]
$ [△] PAGEWIDTH [△] = [△] constant [△] [; comment] $ [△] PW [△] = [△] constant [△] [; comment]
$ [△] TAB [△] = [△] constant [△] [; comment] $ [△] TB [△] = [△] constant [△] [; comment]
$ [△] CAP [△] [; comment] $ [△] CA [△] [; comment]
$ [△] NOCAP [△] [; comment] $ [△] NOCA [△] [; comment]
$ [△] SYMLEN [△] [; comment] $ [△] SL [△] [; comment]
$ [△] NOSYMLEN [△] [; comment] $ [△] NOSL [△] [; comment]

Table 5-2. Control Instructions Not Recognized as Module Header

Control Instruction
$ [△] INCLUDE [△] = [△] file name [△] [; comment] $ [△] IC [△] = [△] file name [△] [; comment]
$ [△] TITLE [△] = [△] 'character string' [△] [; comment] $ [△] TT [△] = [△] 'character string' [△] [; comment]
$ [△] LIST [△] [; comment] $ [△] LI [△] [; comment]
$ [△] NOLIST [△] [; comment] $ [△] NOLI [△] [; comment]
$ [△] EJECT [△] [; comment] $ [△] EJ [△] [; comment]
$ [△] IFDEF △ symbol [△] [; comment]
$ [△] ELSE [△] [; comment]
$ [△] ENDIF [△] [; comment]
$ [△] IF △ expression [△] [; comment]
$ [△] SWITCH △ expression [△] [; comment]
$ [△] CASE △ expression [△] [; comment]
$ [△] DEFAULT [△] [; comment]
$ [△] ENDS [△] [; comment]
$ [△] BREAK [△] [; comment]
$ [△] GENERATE [△] [; comment] $ [△] GEN [△] [; comment]
$ [△] NOGENERATE [△] [; comment] $ [△] NOGEN [△] [; comment]
$ [△] CONDITION [△] [; comment] $ [△] COND [△] [; comment]
$ [△] NOCONDITION [△] [; comment] $ [△] NOCOND [△] [; comment]
$ [△] ERRLOG [△] = [△] 'character string' [△] [; comment] $ [△] EL [△] = [△] 'character string' [△] [; comment]
$ [△] MSGLOG [△] = [△] 'character string' [△] [; comment] $ [△] ML [△] = [△] 'character string' [△] [; comment]
$ [△] IFCHR △ character string 1, character string 2 [, start.end] [△] [; comment]
$ [△] IFSTR △ character string 1, character string 2 [, character string 3, [...]] [△] [; comment]
$ [△] LODM [△] = [△] external macro file name [△] [; comment] $ [△] LM [△] = [△] external macro file name [△] [; comment]

5.3 Purpose of Control Instructions

Table 5-3 List of Control Instructions show the types of the control instructions.

Table 5-3. List of Control Instructions

Type of Control Instruction	Control Instruction
Processor model specification	$PROCESSOR
Mode specification	$MODE

The features of each control instruction are explained below.

Processor model specification control instruction ($PROCESSOR)

(1) Processor model specification control instruction ($PROCESSOR)

[Format]

```
$ [ △ ] PROCESSOR [ △ ] = [ △ ] product name [ △ ] [; comment]
$ [ △ ] PC [ △ ] = [ △ ] product name [ △ ] [; comment]
```

[Purpose]

Specifies the target model of the ST75X in the source module.

[Explanation]

<1> This control instruction is written in the module header of the input source file.

<2> If a product name different from that specified by the C option is used, the specification using the C option takes precedence. At this time, a warning message is output indicating that a different product name has been specified. In the secondary source file, the "$" in the control instruction in the input source file is replaced with a ";" for output, and the product name specified by the C option is output as the processor model specification control instruction.
If a product name same as that specified by the C option is used, a message is not output.

<3> If this control instruction is written twice, an error occurs.

<4> If no product name is specified either by this control instruction or C option, an error occurs.

<5> If this control instruction is written in any part other than the module header, an error occurs.

[Example]

```
$ PROCESSOR = 104
```

Mode specification control instruction ($MODE)

(2) Mode specification control instruction ($MODE)

[Format]

```
$ [ △ ] MODE [ △ ] = [ △ ] constant [ △ ] [; comment]
$ [ △ ] MD [ △ ] = [ △ ] constant [ △ ] [; comment]
```

[Purpose]

Specifies the CPU mode of the 75XL series.

The ST75X only analyzes the format and position of the instruction in the source file.

[Explanation]

<1> This control instruction is written in the module header of the input source file.

<2> If this control instruction is written twice, an error occurs.

<3> If this control instruction is used in any part other than the module header, an error occurs.

<4> If the format of this control instruction is wrong, an error occurs.

[Example]

```
$MODE = 1
```

Remark The input source is not analyzed according to the CPU mode. No corresponding option exists.

CHAPTER 6 PRODUCT OUTLINE

6.1 Product Contents

The ST75X includes the files listed in Table 6-1 Supplied Files.
Correspondingly, the following files are supplied with the "RA75X Assembler Package".

Table 6-1. Supplied Files

File Name	Nature of File
ST75X.EXE ST75X.OM1 ST75X.HLP	Command file Overlay file Help file
STEST1.SRC STEST2.SRC	Sample program file
SRA75X.BAT	Batch file

- The command file is read into the memory first when each program is started.
- The overlay file is read to the memory only when necessary while each program is being executed.
- The sample program file is used to check the operation of the structured assembler.
- Place ST75X.OM1 in the current drive when MS-DOS V2.11 is used.
- ★ The above files must be in the same directory.

6.2 System Configuration

The host machines and OSs with which the structured assembler can be used are as follows:

- PC-9800 series (MS-DOS™)
- IBM PC/AT™ (PC DOS™)

For more details on the host machine and OS, refer to **RA75X Assembler Package User's Manual - Operation**.

★ 6.3 Device Files

A device file is necessary for the 75X series.
The device file is searched for in the following sequence:

<1> Path specified by Y option
<2> "..\dev" (relative path to path in which ST75X is started)
<3> Path where ST75X was started
<4> Current path
<5> Path specified by environmental variable "PATH"

[MEMO]

CHAPTER 7 OPERATION

7.1 Structured Assembler I/O Files

Table 7-1 Structured Assembler I/O Files lists the I/O files of the structured assembler (ST75X).

Table 7-1. Structured Assembler I/O Files

	Type of File	Default File Type
Input file	Source module file Source module file written in structured assembly language	—
	Parameter file[Note] Options specified on starting structured assembler are created in advance as parameter file.	.PST
Output file	Secondary source module file Source module file translated into assembly language of RA75X	.ASM
	Error list file File containing error information on structured assembly	.EST

Note Refer to **7.3.1 (2) Starting with parameter file.**

7.2 Purpose of Structured Assembler

The structured assembler reads a source module file and translates the structured assembly language into an assembly language.

If an error is found in the source module, an error message is output to the error list.

The structured assembler performs processing in accordance with options specified on starting. For the more information on the options, refer to **7.4 Structured Assembler Options.**

The maximum limits of the structured assembler are shown below.

★

Item	Maximum Value
Length of line	254 characters (excluding LF and CR)
Number of symbols that can be used	512 (except reserved words)
Nesting level of control statements	31 levels
Nesting level of conditional processing instructions	8 levels
Number of #defgeti instructions that can be used	48
Number of operands that can be successively assigned	33

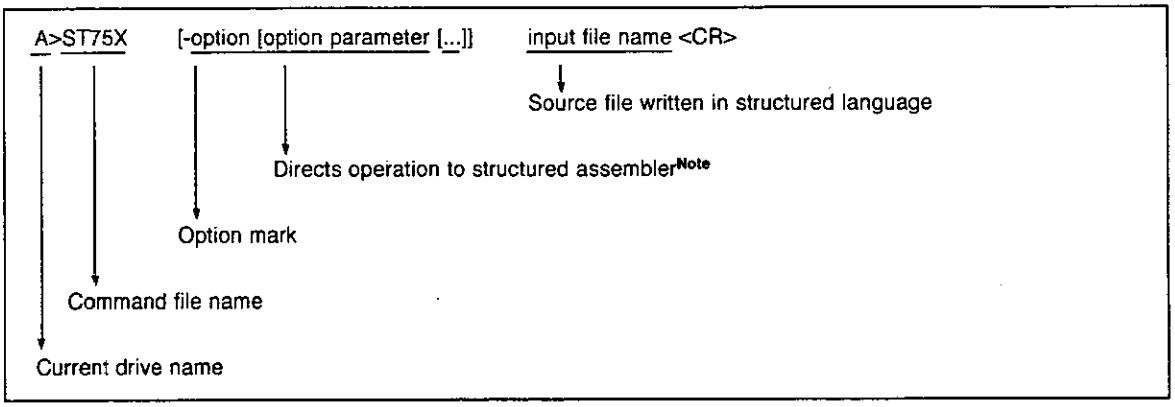
7.3 Starting the Structured Assembler

7.3.1 Starting the structured assembler

The structured assembler can be started in two ways.

(1) Starting from the command line

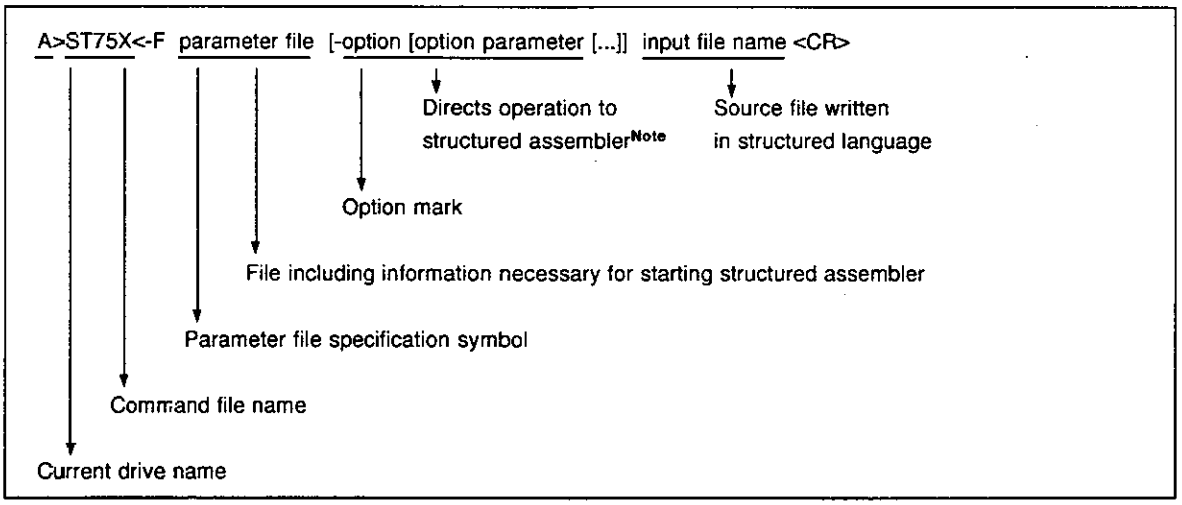
Input the following command to start the structured assembler.



Note To specify two or more options, delimit each option from the others by a blank.

(2) Starting with parameter file

It is sometimes inconvenient to have to specify the same parameter again and again every time you start the structured assembler. In this case, start the structured assembler using a parameter file.



Note To specify two or more options, delimit each option from the others by a blank.

- The parameter file is created with the editor.
- Write the structured assembler options in the parameter file.
- To change or add to the options specified in the parameter file, structured assembler options can also be specified at the command line after the parameter file name. If conflicting options of the same type are specified in the parameter file and at the command line, the option specified at the command line takes precedence.
- ★ • Nesting cannot be performed in the parameter file.
- ★ • In a parameter file, a character string starting with ";" or "#" and ending with LF or EOF is interpreted as a comment.
- ★ • When specifying an input file name in the parameter file, do not use the input file name at the command line. If input file names are specified twice, an error occurs.

Example 1. To start the structured assembler by the specifying parameter file 'ST.JOB'

- Contents of 'ST.JOB'

```
-WT4, 5, 6 -EB: \RA75X\ERROR.EST
```

- Starting with parameter file

```
A>ST75X TEST.SRC -C064 -FST.JOB

Structured assembler preprocessor for RA75X VX.XX [dd Mmm yy]
   Copyright (C) NEC Corporation 1988,1995

start

 Target chip : uPD75XXXX
 Device file : VX.XX

Conversion complete, 0 error(s) found.

A>
```

Example 2. To change or add to options specified in the parameter file at the command line

```
A>ST75X TEST.SRC -C064 -FST.JOB -OB: \RA75X\SAMPLE.ASM
. . . . . .
A>
```

7.3.2 Execution start and end messages

(1) Execution start message

When the structured assembler is started, the following execution start message is displayed on the console.

```
Structured assembler preprocessor for RA75X VX.XX [dd Mmm yy]
  Copyright (C) NEC Corporation 1988,1995
```

(2) Processing display message

```
start
```

A full-stop "." is displayed each time 100 lines have been processed.

(3) Execution end message

- If no error is detected, the following message is output to the console and control is transferred to the OS.

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

- If an error or errors are found, the number of errors is output to the console, and control is transferred to the OS.

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

- If a fatal error that makes continuation of processing impossible is detected, the following message is output to the console, the processing is aborted, and control is transferred to the OS.

```
A/ST75X TEST.XXX -CXXX

Structured assembler preprocessor for RA75X VX.XX [dd Mmm yy]
   Copyright (C) NEC Corporation 1988,1995

start
TEST.XXX(2) : F227 Illegal operand in a line
Conversion complete, 1 error(s) found.

 Target chip : uPD75XXXX
 Device file : VX.XX

A>
```

7.4 Structured Assembler Options

7.4.1 Type of structured assembler option

The structured assembler options give detailed directions on the operation of the structured assembler. The options listed in Table 7-2 Types of Structured Assembler Options are available.

★ **Table 7-2. Types of Structured Assembler Options**

Option	Option Name	Format	Default Assumption on Omission of Option
C	Model specification	-C model name	Must not be omitted.
D	Symbol definition specification	-D symbol [=numeric value]	Symbol = 1
WT	Number of tabs specification	-WT numeric value 1, numeric value 2, numeric value 3	numeric value 1 = 2, numeric value 2 = 3, numeric value 3 = 4
I	Include file path specification	-I [drive number:] directory	It is assumed that current drive and current directory are specified.
O	Secondary source file specification	-O [drive number:] [directory] output file name	File replacing file type of input file with ".ASM" is created in current directory.
E	Error list file specification	-E [drive number:] [directory] output file name	File replacing file type of input file with ".EST" is created in current directory.
F	Parameter file specification	-F [drive number:] [directory] output file name	If file type of input file is omitted, ".PST" is assumed.
J	Secondary source file forced output specification	-J	—
M	Mode specification	-M mode name	—
S, NS	Symbol name length specification	-S, -NS	31 characters
GS, NGS	Debug information output specification	-GS, -NGS	Output
Y	Device file search path	-Y [drive number:] directory	—
-	Help specification	- -	—

—: None

7.4.2 Specifying option

(1) Option mark

An option mark can be chosen freely by assigning a character to environmental variable "OPTMARK".
The default option mark character is "-".
If two or more characters are assigned to environmental variable "OPTMARK", the first one character is valid.

(2) Option name

Uppercase and lowercase characters are not distinguished in an option name.
Write option names after an option mark, without any blanks.

(3) Option specification position

An option can be specified at any place, before or after the input file.

(4) Option parameter

Write option parameters after an option without any blanks.

(5) Duplicate option specification

If options of the same name are specified twice, the option specified last is valid.

7.4.3 Explanation of options

In the section, the options are explained in detail.

C

(1) Model specification (C option)

[Format]

```
-C model name
```

[Purpose]

Specifies the target model of the structured assembler.

[Explanation]

<1> Creates the assembler best suited for the model name specified after "-C".

★ <2> If a model name is not specified by the processor model specification control instruction (PROCESSOR, PC instruction) in the input source file, and if no model is specified by this option, an error occurs.

★ <3> Use alphanumeric characters to specify the model name. Uppercase and lowercase characters are not distinguished.

[Example]

```
A>ST75X TEST.SRC -C0008
```

[Model Name List]

The target devices in the 75X and 75XL series and model names are listed in the tables below.

<75XL series>

- **Extended high-end**

Model Name	Target Device	Model Name	Target Device
117H	μPD75117H, 75P117H	238	μPD75238, 75P238
217	μPD75217	517	μPD75517
218	μPD75218, 75P218	518	μPD75518, 75P518
236	μPD75236	617A	μPD75617A
237	μPD75237		

- **High-end**

Model Name	Target Device	Model Name	Target Device
104	μPD75104, 75104A	208	μPD75208
106	μPD75106	CG208	μPD75CG208
108	μPD75108, 75108F, 75108A, 75P108B	212A	μPD75212A
		216A	μPD75216A, 75P216A
P108	μPD75P108	CG216	μPD75CG216A
112	μPD75112, 75112F	336	μPD75336, 75P336
116	μPD75116, 75116F, 75P116	352A	μPD75352A
116H	μPD75116H	512	μPD75512
206	μPD75206	516	μPD75516, 75P516

C

- **Standard**

Model Name	Target Device	Model Name	Target Device
004	μPD75004	304	μPD75304, 75304B
006	μPD75006	306	μPD75306, 75306B
008	μPD75008, 75P008	308	μPD75308, 75308B, 75P308
028	μPD75028	312	μPD75312
036	μPD75036, 75P036	312B	μPD75312B
048	μPD75048, 75P048	316	μPD75316, 75P316
064	μPD75064	316A	μPD75P316A
066	μPD75066	316B	μPD75316B, 75P316B
068	μPD75068, 75P068	328	μPD75328, 75P328
268	μPD75268		

Caution The structured assembler does not support the low-end devices of the 75XL series (μPD75P402 and 75402A).

★ <75XL series>

Model Name	Target Device	Model Name	Target Device
0004	μPD750004	3036	μPD753036
0006	μPD750006	P3036	μPD75P3036
0008	μPD750008	3104	μPD753104
P0016	μPD75P0016	3106	μPD753106
0064	μPD750064	3108	μPD753108
0066	μPD750066	P3116	μPD75P3116
0068	μPD750068	3204	μPD753204
P0076	μPD75P0076	3206	μPD753206
0104	μPD750104	3208	μPD753208
0106	μPD750106	P3216	μPD75P3216
0108	μPD750108	3304	μPD753304[Note]
P0116	μPD75P0116	4202	μPD754202
3012	μPD753012	4144	μPD754144
3012A	μPD753012A	4244	μPD754244
3016	μPD753016	4264	μPD754264
3016A	μPD753016A	F4264	μPD75F4264[Note]
3017	μPD753017	4302	μPD754302
3017A	μPD753017A	4304	μPD754304
P3018	μPD75P3018	P4308	μPD75P4308
P3018A	μPD75P3018A		

Note Under development

D

(2) Symbol definition (D option)

[Format]

```
-D symbol [= numeric value]
```

[Purpose]

Defines a symbol.

★ **[Explanation]**

<1> This option defines a symbol. The value given to a symbol using this option can be a binary, octal, decimal, or hexadecimal number. If no value is specified, 1 is assumed.

<2> This option has the same effect as defining a symbol by using the symbol definition pseudoinstruction (define). If a reserved word is specified as a symbol, however, an error occurs.

<3> Up to 30 symbols can be defined with each delimited by a comma from the others on the start line. The same symbol name can be defined in duplicate up to 30 times.

<4> If a symbol defined by the symbol definition pseudoinstruction (define) is defined again by this option, a warning message is output, and the definition by the symbol definition pseudoinstruction (define) takes precedence.

<5> Uppercase and lowercase characters are not distinguished.

[Example]

<TEST.SRC>

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

<Starting method>

```
A>ST75X TEST.SRC -DTRUE=1 -C064
```

<TEST.ASM>

```
Text 1
```

WT

(3) Number of tabs setting (WT option)

[Format]

-WT numeric value 1, numeric value 2, numeric value 3

[Purpose]

Sets the number of tabs until the translated assembly language is output.

★ **[Explanation]**

<1> numeric value 1 specifies the number of tabs until an instruction is output (default = 2).
numeric value 2 specifies the number of tabs until the operand of the instruction is output (default = 3).
numeric value 3 specifies the number of tabs until a comment is output (default = 4).

<2> Specify such that numeric value 1 < numeric value 2 < numeric value 3.

<3> Input numeric values as decimal numbers. The ranges in which numeric values 1 through 3 can be specified are as follows:

- 0 ≤ numeric value 1 ≤ 97
- 1 ≤ numeric value 2 ≤ 98
- 2 ≤ numeric value 3 ≤ 99

<4> If an illegal value is used for any of numeric values 1 through 3, an error occurs.

[Example]

<TEST.SRC>

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

<Starting method>

```
A>ST75X TEST.SRC -WR3, 4, 5 -C064
```

<TEST.ASM>

```
                        MOV     A,#0        ;A = #0
                        SKE     A,#1H       ;if (A == #1H)
                        BR      ?L1
                        MOV     XA,#0C5H    ;     XA = #0C5H
?L1:                                        ;endif
      *.........*.........*.........*.........*.........*
                        3       4           5           Number of tabs
```

I

(4) Include file specification (I option)

[Format]

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

[Purpose]

Specifies an include file that is input to the structured assembler.

[Explanation]

<1> This option specifies the drive number and directory of the include file.

<2> If this option is omitted, it is assumed that the file is in the current drive and current directory.

[Description Example]

If TEST.SRC is in the same drive as ST75X.EXE, and if FILE is in directory INCLUDE of drive B

<TEST.SRC>

```
#include  "FILE"

        A=SIZE1
        B=SIZE2
```

<FILE>

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

<Starting method>

```
A>ST75X TEST.SRC IB:INCLUDE
```

<TEST.ASM>

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

O

(5) Secondary source file specification (O option)

[Format]

```
-O [drive number:] [directory] output file name
```

[Purpose]

Specifies the output destination of the output file and a file name.

[Explanation]

<1> This option specifies the output drive number, directory, and file name of the secondary source file after translation.

<2> If this option is omitted, a file with the same name as the input file but with the file type changed to "ASM" is created in the current directory as an output file.

<3> If no output file name is specified when this option is specified, the output file name is the same as that of the input file but with the file type is changed to ".ASM".

★ <4> When an error occurs, the secondary source file is not created.

<5> 'NUL' or 'AUX' can be specified as a file name.

[Example of Use]

<TEST.SRC>

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

<Starting method>

```
A>ST75X TEST.SRC -OSAMPLE.ASM -C064
```

<SAMPLE.ASM>

```
                SKE     A,#1H    ;if (A == #1H)
                BR      ?L1
                MOV     XA,#0C5H ;    XA = #0C5H
?L1:                             ;endif
```

E

(6) Error list file specification (E option)

[Format]

```
-E [drive number:] [directory] output file name
```

[Purpose]

Specifies the output destination of the error list file and a file name.

[Explanation]

<1> This option specifies the output drive number, directory, and file name of the error list file.

<2> If this option is omitted, a file with the same name as the input file but with the file type is changed to "EST" is created in the current directory as the error list file.

<3> If an error list file name is omitted when this option is specified, a file with the same name as the input file but with the file type is changed to "EST" is used as an error list file.

<4> The error list file is not created on normal completion.

<5> 'NUL' or 'AUX' can be specified as a file name.

[Example of Use]

<TEST.SRC>

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

<Starting method>

```
A>ST75X TEST.SRC -ESAMPLE.EST -C064
.....
start
TEST.SRC (2)  :F221 Missing ENDIF
Conversion complete, 1 error (s) found.

A>
```

<SAMPLE.EST>

```
TEST.SRC (2)  :F221 Missing ENDIF
Conversion complete, 1 error (s)  found.
```

F

(7) Parameter file specification (F option)

[Format]

```
-F [drive number:] [directory] input file name
```

[Purpose]

Specifies the input destination of a parameter file and file name (refer to **7.3.1 (2) Starting with parameter file**).

[Explanation]

<1> This option specifies the input drive number, directory, and file name of the parameter file.
<2> If no parameter file name is specified when this option is used, an error occurs.
<3> If no file type is specified when a parameter file is specified, file type "PST" is assumed.
<4> 'NUL' can be specified as a file name.

[Example of Use]

<TEST.SRC>

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

<SAMPLE.PST>

```
TEST.SRC -ESAMPLE.EST
```

<Starting method>

```
A>ST75X -FSAMPLE.PST -C064
.....................................
start
Conversion complete, 0 error (s) found.

A>
```

As a result, TEST.SRC is subject to processing by the structured assembler, and the secondary source file is created in TEST.ASM. If an error occurs, an error file is created in SAMPLE.EST.

<TEST.ASM>

```
                SKE     A,#1H    ; if (A == #1H)
                BR      ?L1
                MOV     XA,#0C5H ;      XA = #0C5H
?L1:                             ; endif
```

<SAMPLE.EST>

```
TEST.SRC (2) :F221 Missing ENDIF.
Conversion complete, 1 error (s) found.
```

J

(8) Secondary source file forced output specification (J option)

[Format]

```
-J
```

[Purpose]

Forcibly outputs the secondary source file.

[Explanation]

<1> This option outputs the secondary source file on completion due to a fatal error.

<2> The input source is output as is on the error line.

M

★ **(9) Mode specification (M option)**

[Format]

-M mode name mode name: 1 or 2

[Purpose]

Specifies the mode of the target model of the structured assembler.

[Explanation]

<1> Either MkI mode or MkII mode is specified as the mode.
Input "1" for MkI mode, and "2" for MkII mode.

<2> Specify a mode only when the target model is the 75XL series.
If a mode is specified when the target model is the 75X series, or if no mode is specified when the target model is the 75XL series, an error occurs.

<3> If no mode is specified by the mode specification control instruction (MODE, MD instruction) in the input source file, the mode specified by this option is output to the secondary source file as a mode specification control instruction.

S, NS

★ **(10) Symbol name length specification (S and NS options)**

[Format]

```
-S
-NS
```

[Purpose]

Specifies the valid number of characters of a symbol.

- -S ... Up to 31 characters
- -NS ... Up to 8 characters

[Explanation]

<1> As default assumption, the valid number of characters of a symbol is up to 31.

<2> The symbols affected by this option are the symbols defined by the symbol definition pseudoinstruction (define), user symbols, or symbols referenced by the conditional pseudoinstruction (indef/else/endif).

<3> When the S option is specified, and when "$NOSYMLEN" is written in the input source file, "$SYMLEN" is output to the secondary source file.

<4> When the NS option is specified, and when "$SYMLEN" is written in the input source file, "$NOSYMLEN" is output to the secondary source file.

<5> If the S option and NS option are specified in duplicate, the option specified latter takes precedence.

GS, NGS

★ **(11) Debug information output specification (GS and NGS options)**

[Format]

```
-GS
-NGS
```

[Purpose]

Specifies whether debug information is output to the secondary source file.

- -GS ... Outputs debug information.
- -NGS ... Does not output debug information.

[Explanation]

<1> As default assumption, the debug information is output.

<2> The GS option replaces the first character "$" with ";" for output if the input source file has debug information.

Example

$DGS~ → ; DGS~
$DGL~ → ; DGL~

<3> When the GS option is specified, or in the default status, "$NODEBUGA" is output to the secondary source file.

<4> If the GS and NGS options are specified in duplicate, the option specified latter takes precedence.

Cautions 1. When the GS option is specified or in the default status with the ST75X, do not specify the GA and NGA options with the RA75X.

2. The IE-75000-R and IE-75001-R cannot perform source debugging.

Y

★ **(12) Device file search path specification (Y option)**

[Format]

-Y [drive number:] directory

[Purpose]

Specifies a path from which the device file is read.

[Explanation]

<1> If a name other than a path name is specified, or if a path name is omitted, an error occurs.

<2> If directory specification symbol "\" is not included at the end of the directory, the ST75X assumes that the symbol is there.

<3> When this option is not specified, the device file is searched in the following sequence:

(a) Path specified by Y option

(b) "..\dev" (relative path to path where ST75X was started)

(c) Path where ST75X was started

(d) Current path

(e) Path specified by environmental variable "PATH"

★ **(13) Help specification (- option)**

[Format]

```
--
```

[Purpose]

Displays the contents of the help file (ST75X.HLP).

★ **[Explanation]**

The contents of the help file are shown below.

```
Usage : st75x [option[...]] input-file [option[...]]

The option is as follows ([] means omissible, ... means repetition).
  -cx          :Select target chip.  (x = 004, 104, etc.) *Must be specified.
  -o[file]     :Create the assembler source file [with the specified name].
  -e[file]     :Create the error list file [with the specified name].
  -ffile       :Input options or source file name from specified file.
  -idirectory :Set include search path.
  -s/-ns       :Expand symbol length up to 31/or symbol length in 8.
  -wtn1/-wt[n1],n2/-wt[n1],[n2],n3
               :Specify the number of tabs up to output position of each field.
                 n1:Output position mnemonic field.
                 n2:Output position operand field.
                 n3:Output position comment field.
                 *Must be 0 <= n1 < n2 < n3 < 100
  -dname[=data][,name[=data][...]]
               :Define name [with data].
  -j           :Create the assembler source file if fatal error occurred.
  -gs/-ngs     :Output the structured assembler source debug information to
                assembler source file/Not.
  -y           :Set device file search path.
  --           :Show this message.
DEFAULT ASSIGNMENT:       -o -e -wt2,3,4 -gs
```

★ 7.5 Setting Options from Project Manager

The options of the structured assembler can be specified from the project manager.
For more information on the project manager, refer to **Project Manager User's Manual - Reference.**

Cautions 1. When the dependency of the include files is checked while a make file is created, only comments and character strings are deleted, and conditions such as #if and #_if are ignored.

Example
```
#ifdefSYM
#include "func1.inc"
#else
#include "func2.inc"
#endif
```

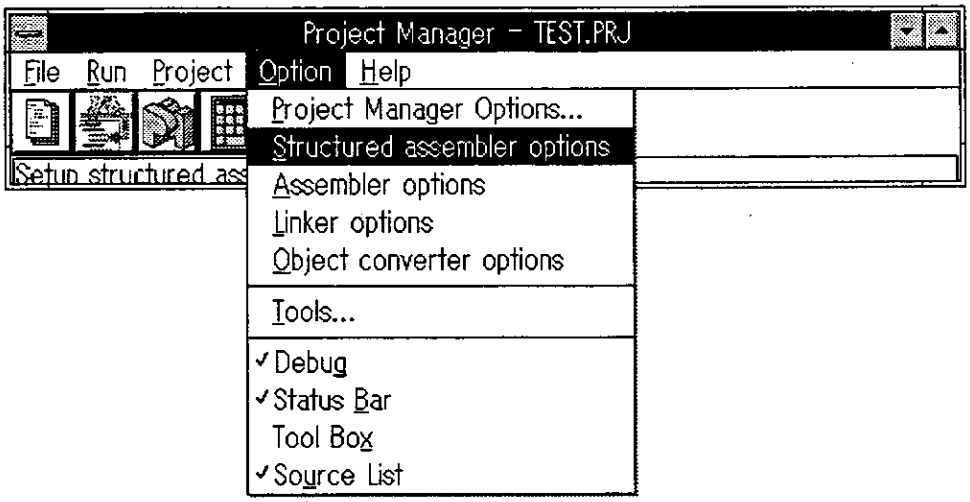
In this example, the lines #ifdef, #else, and #endif are ignored; therefore, both the files are interpreted as include files. If these files do not exist, an error occurs on building, regardless of whether these files are actually referenced.

2. **The structured assembler is normally started with the assembler. When options are set from the project manager at this time, the options C, Y, and NGA are appended to the assembler. The user cannot add the C option at this time.**
3. **The [Option] → [Debug] menu in the project manager is ignored. Specify debug information by using the GS option in the [Structured assembler options] menu or GA option in the [Assembler options] menu.**

7.5.1 Option menu item

The option menu items that can be set from the project manager are shown below.

Figure 7-1. Option Setting Menu



7.5.2 Option setting dialog box

The dialog box for setting the options of the structured assembler preprocessor is explained below.

Select [Option] → [Structured assembler options], or [Option] → [Source List] from the menu, and select the "Option" button in the dialog box. The structured assembler option setting dialog box will be opened.

Figure 7-2. Option Setting Dialog Box (if source file is not selected)

... Sets options for all source files

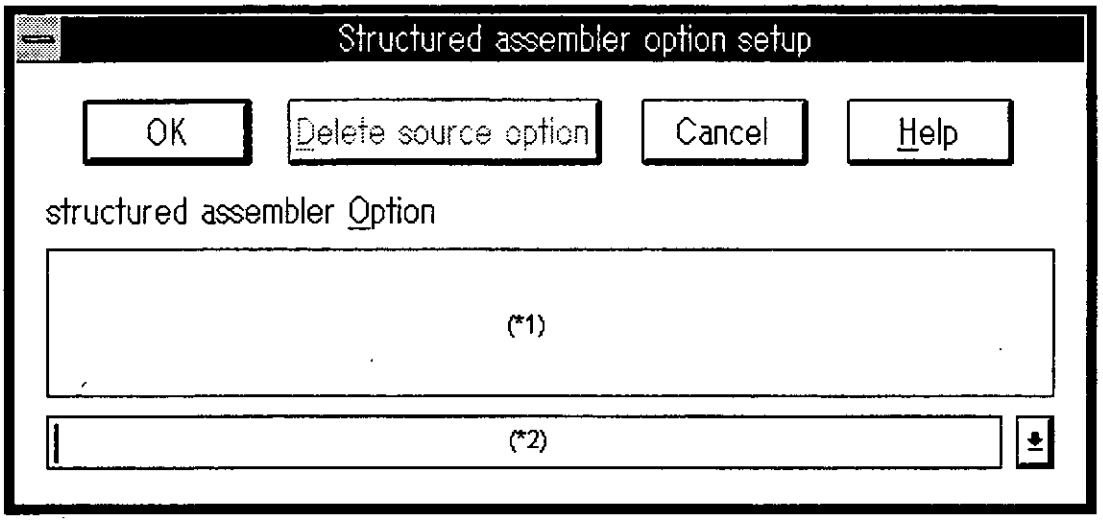


Figure 7-3. Option Setting Dialog Box (if source file is selected)

... Sets options for selected source file

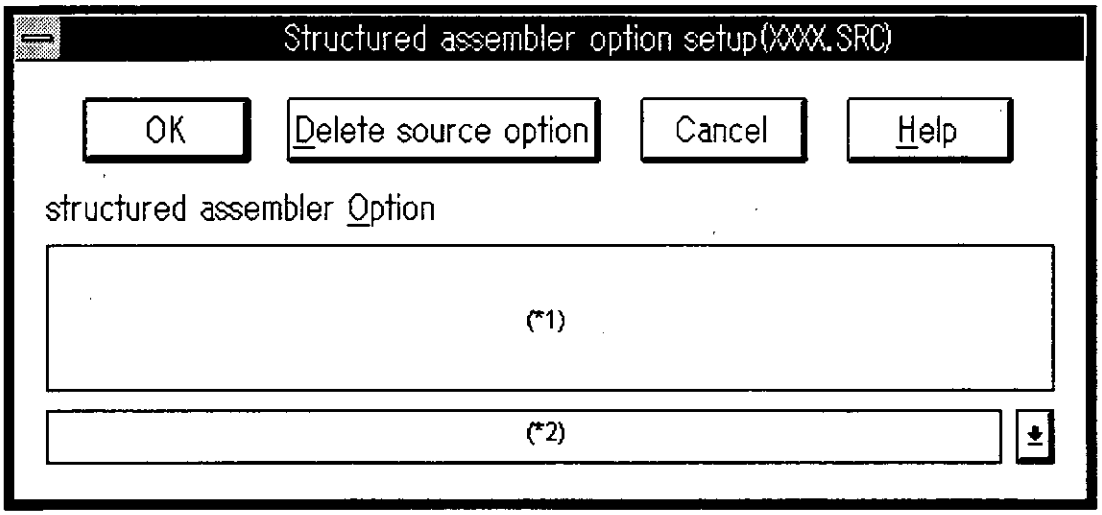


Table 7-3. Features of Option Setting Dialog Box

Button/Box	Explanation
"OK" button	If a source file is not selected, sets options for all source files (source files for which individual options are not set), and closes the dialog box. If a source file is selected, sets options for the selected source file, and closes the dialog box. If the return key is pressed when the focus is in the option input combo box, it is assumed that the "OK" button has been pressed.
"Delete source option" button	Valid only when a source file is selected. When this button is selected, the individual options set in source file units are deleted. For a source file from which the individual options have been deleted, the All option becomes valid.
"Cancel" button	Clears setting of this dialog box, and closes the dialog box. When the ESC key is pressed, it is assumed that the "Cancel" button have been pressed.
"Help" button	Opens the help file related to this dialog box.
Option character string display area (*1)	Displays the character string of the option currently set. Option character strings that do not fit in one line can also be displayed. The option character string input to the option input combo box is displayed in this area as it is input.
Option input combo box (*2)	Used to input an option character string Up to 127 characters can be used[Note]. A device file does not need to be specified in this box because it is specified in the project manager.
[↓]	Shows a drop-down input history. Up to 10 entries are saved (10 options without source, and 10 options each for a specific source). If a character string same as an input option character string in the history, the input in the past history deleted, and the newly input character string is added.

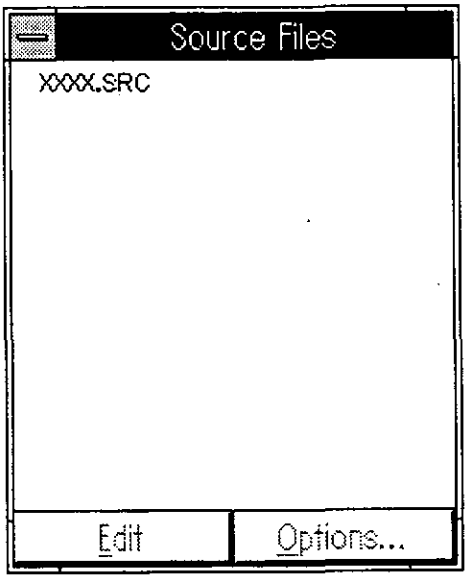
Note Including the number of characters of the source file name and option automatically set by the project manager.

Caution The options are not checked for error when they are set. If there is an error in the option input, an error occurs on building.

7.5.3 Source file option setting dialog

When [Option] → [Source List] is selected from the menu, a dialog box will be opened (refer to **Figure 7-4**).

Figure 7-4. [Source List] Dialog Box



When a source file is selected and the [Option] button is pressed, the assembler option setting dialog box is opened (refer to Figure 7-3). When an option is input to the option input combo box (*2) and the "OK" button is pressed, the source file option setting dialog box is opened (refer to **Figure 7-5**).

Figure 7-5. Source File Option Setting Dialog Box

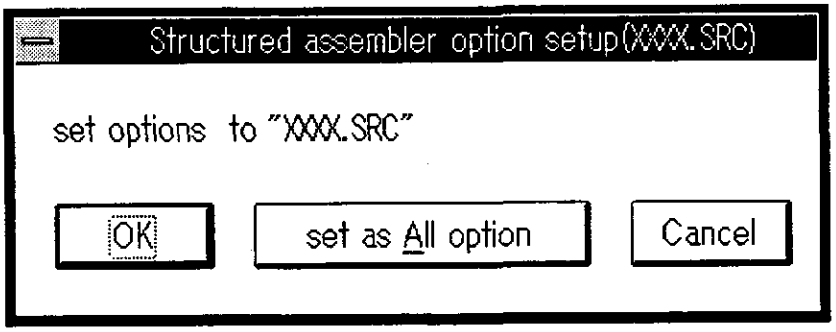


Table 7-4. Features of Source File Option Setting Dialog Box

Button	Explanation
"OK" button	Sets options for selected source file, and closes the dialog box.
"set as All option" button	Sets options for all source files (source files for which individual options are not set).
"Cancel" button	Clears setting of this dialog box, and closes the dialog box.

[MEMO]

CHAPTER 8 I/O FILES

The source module files of the input files of the structured assembler are divided into the following two types:

- Input source program file
- Include file

The output files are classified into the following two types:

- Secondary source program file
- Error list file

8.1 Input Source Program File

The source program files input to the structured assembler contain assembly language within the structured assembly language code subject to processing by the structured assembler.

Figure 8-1 shows an example of an input source program file.

Figure 8-1. Input Source Program Example

```
if (B == #0)
    TMOD0 = XA
    XA = #0CH
else
    XA = #0AH
elseif
    TMO = XA
CALL    ! XXX
```

8.2 Include File

8.2.1 What is an include file?

The contents of files other than the input source program file can be included in the source program file. These other files are called include files.

By storing versatile subroutines in the form of include files, they can be used in a variety of input source programs.

8.2.2 Using include files

An include file is expanded at the position of the #include pseudoinstruction written in the source program.

Figure 8-2 shows an example of an input source program including an #include pseudoinstruction and include file.

Figure 8-2. Example of an Include File

<Input source file>

```
#include    "FILE"

    A=SIZE1
    B=SIZE2
```

<Include file> (FILE)

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

<Output source file>

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

8.3 Secondary Source Program Files

A secondary source program file is a source program file output by the structured assembler, and is used as an input source program by the assembler.

The secondary source program file is created as follows:

Input Source Program File	Secondary Source Program File
RA75X assembly language	Output as is
Control statements in structured assembler Expressions in structured assembler	Output as comments, or statements converted RA75X assembly language.
Comments in structured assembler	Output as is
Pseudoinstructions in structured assembler	Not output

Figure 8-3 shows an example of how the secondary source program file is created.

Figure 8-3. Example of Source Program

<Input source file>

```
#include     "FILE"
    if( B == #0)
            TMODO = #0CH ( XA )
    elseif
    CALL     !XXX
    A = SIZE1
    B = SIZE2
```

<Include file> (FILE)

```
#define      SIZE1    08H
#define      SIZE1    08H
```

<Output source file>

```
                SKE     B,#0        ;if( B == #0)
                BR      ?L1
                MOV     XA,#0CH     ;   TMODO = #0CH ( XA )
                MOV     TMODO,XA
?L1:                                ;elseif
                CALL    !XXX        ;CALL   !XXX
                MOV     A,08H       ;A=SIZE1
                MOV     B,0AH       ;B=SIZE2
```

8.4 Error List File

An error list file stores the error messages output when the structured assembler is run. Figure 8-4 shows an example of an error list file.

Figure 8-4. Example of Error List File

<TEST.SRC>

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

<Starting method>

```
A>ST75X TEST.SRC -ESAMPLE.EST -C064
.....................................
start
TEST.SRC(2) :F221 Missing ENDIF
Conversion complete, 1 error(s) found.

A>
```

<TEST.EST>

```
TEST.SRC(2) :F221 Missing ENDIF
Conversion complete, 1 error(s) found.
```

[MEMO]

9.1 Structured Assembly and Assembly

Assembly using the structured assembler and normal assembler can be executed together using batch processing in MS-DOS.

This chapter explains how to execute structured assembly and assembly together using the batch file (SRA75X.BAT) and test program (STEST1.SRC) supplied with the package.

9.1.1 Outline of SRA75X.BAT

The batch file SRA75X.BAT is used to execute the structured assembler and assembler in succession. The contents of this file are shown below.

<SR75X.BAT>

```
ST75X %1.SRC -C%2
ECHO OFF
IF ERRORLEVEL 1 GOTO END
RA75X %1.ASM %3 %4 %5 %6 %7 %8 %9
:END
```

This SRA75X.BAT is a batch file that inputs a file with file type SRC and creates a load module file with file type REL (the file name of the output file is the file name of the input file). If an error or fatal error occurs while the structured assembler is being executed, batch processing is aborted without assembler processing being executed.

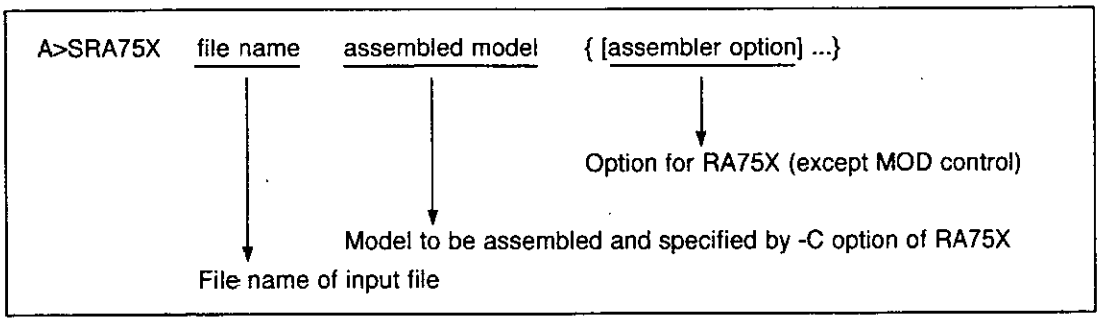
Caution When specifying options for the structured assembler, write the options in advance in the batch file.

Example <SRA75X.BAT>

```
ST75X %1.SRC -OSAMPLE.ASM
ECHO OFF
IF ERRORLEVEL 1 GOTO END
RA75X %1.ASM %3 %4 %5 %6 %7 %8 %9
:END
```

9.1.2 Inputting commands

Input commands as shown below.



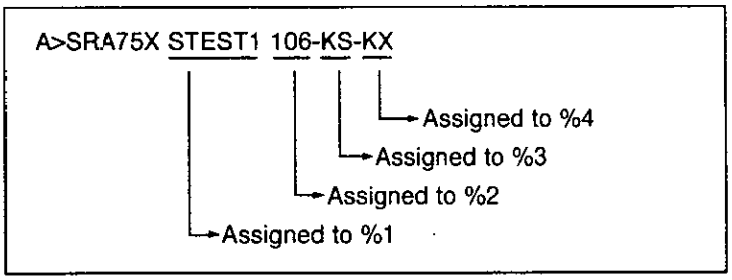
Caution A character string including "=" cannot be specified as an assembler option for the batch file. To specify a character string including "=" as an assembler option, write it directly in the batch file.

9.1.3 Operation

The execution of SRA75X.BAT is explained below, using a test program (STEST1.SRC).

(1) Input a command.

★



(2) SRA75X.BAT displays the following information on the console, and terminates processing.

(a) If structured assembler finds no errors

```
A>SRA75X STEST1 106 -KS -KX

A>ST75X STEST1.SRC -C106

Structured assembler preprocessor for RA75X VX.XX [dd Mmm yy]
      Copyright (C) NEC Corporation 1988,1995

start
Conversion complete, 0 error(s) found.

A>ECHO OFF
75X Series Assembler VX.XX [dd Mmm yy]
   Copyright (C) NEC Corporation 1985

ASSEMBLY START

 TARGET CHIP : UPD75106
 STACK SIZE = 000AH

 ASSEMBLY COMPLETE,  NO ERROR FOUND
A>
```

In this case, a print file including a symbol table list and cross reference list is created by specifying KS and KX.

(b) If structured assembler outputs a (fatal) error

```
A>ST75X STEST1.SRC -C106

Structured assembler preprocessor for RA75X VX.XX [dd Mmm yy]
   Copyright (C) NEC Corporation 1988,1995

A001 Missing input file

Program aborted

A>ECHO OFF
A>
```

In this case, the batch processing is terminated without executing the assemble processing. Correct the errors in STEST1.SRC and execute the batch processing again.

9.2 Example of Structured Assembler Program

In order to use the structured assembler effectively, please refer to the **75X Series Structured Assembler Processor Application Note** available.

CHAPTER 10 ERROR MESSAGES AND TERMINATION PROCESSING INFORMATION

10.1 Error Messages

10.1.1 Abort errors

An abort error message is output on starting, or if execution cannot be continued. After the error message has been output, execution of the program is immediately stopped, and control is returned to the OS.

(1/3)

Message	A001 Missing input file
Cause	Only options other than -F and – – are specified and an input file is not specified, or, a help file that is started only by specifying an execution program name does not exist.
User action	Correctly specify the input file name.
Message	A002 Too many input files
Cause	Two or more input files are specified.
User action	Specify only one input file.
Message	A004 Illegal file name 'file name'
Cause	The type, characters, or number of characters of the input file name is wrong.
User action	Specify the file name with the correct type, characters, and number of characters.
Message	A005 Illegal file specification 'file name'
Cause	An illegal file is specified.
User action	Specify the correct file.
Message	A006 File not found 'file name'
Cause	The specified input file does not exist.
User action	Specify a file name that exists.
Message	A008 File specification conflicted 'file name'
Cause	I/O file names are specified in duplicate.
User action	Specify different names for input and output files.
Message	A009 Unable to make file 'file name'
Cause	The specified file is write-protected.
User action	Release the file from write protection.
Message	A010 Directory not found 'file name'
Cause	A drive or directory that does not exist is included in the specified file name.
User action	Check the drive or directory, and specify a correct file.
Message	A011 Illegal path 'option'
Cause	A wrong path name is specified for an option that specifies a path.
User action	Specify a correct path.
Message	A012 Missing parameter 'option'
Cause	A necessary option parameter is not specified.
User action	Specify an option parameter.

(2/3)

Message	A014 Out of range 'option'
Cause	The numeric value specified by the option is out of range.
User action	Specify a numeric value within the range.
Message	A015 Parameter is too long 'option'
Cause	The number of characters of the option parameter exceeds the limit.
User action	Keep the number of characters of the option parameter to within the limit.
Message	A016 Illegal parameter 'option'
Cause	The syntax of the option parameter is wrong.
User action	Specify the option parameter in the correct syntax.
Message	A017 Too many parameters 'option'
Cause	The total number of option parameters exceeds the limit.
User action	Keep the number of option parameters to within the limit.
Message	A018 Option is not recognized 'option'
Cause	The specified option name is wrong.
User action	Specify a correct option name.
Message	A019 Parameter file nested
Cause	The -F option is specified in the parameter file.
User action	Do not specify the -F option in the parameter file. Specify it only on the command line.
Message	A020 Parameter file read error 'file name'
Cause	Incorrect code exists in the parameter file.
User action	Specify a correct parameter file.
Message	A021 Memory allocation failed
Cause	The memory capacity is insufficient.
User action	Allocate the necessary memory capacity.
Message	A101 Open/read/write/close error on 'file name'
Cause	The file cannot be correctly opened, read, written, or closed because an error occurs when the file is input/output.
User action	Check the file (if the file is protected, release the protection).
Message	A102 Can't find 'file name'
Cause	The include file is missing, or its name is the same as an input file name or output file name.
User action	Specify the correct path, directory, and file.
Message	A103 Illegal include file 'file name'
Cause	An illegal file is specified for the include file.
User action	Specify the correct file.
Message	A105 Can't define the reserved symbol
Cause	A reserved word is specified with the -D option.
User action	Do not specify a reserved word with the -D option.
Message	A106 Duplicate PROCESSOR control
Cause	The PROCESSOR control instruction is specified in duplicate in the source file. Or, a model different from that specified by the -C option is specified.
User action	Specify only one PROCESSOR control instruction, or correct the model name.

(3/3)

Message	A107 No processor specified
Cause	A target model name is not specified.
User action	Specify a target model name.
Message	A108 Illegal processor type specified
Cause	A model name other than target model names is specified by the PROCESSOR control instruction in the source file.
User action	Specify a correct target model name.
Message	A109 Illegal processor type specified -C 'model'
Cause	A model name other than target model is specified by model specification option (-C).
User action	Specify a correct model name.
Message	A110 Can't use this control outside module header
Cause	A control instruction that must be specified in the source module header is specified on the normal source line.
User action	Specify the control instruction specified on the source line in the source module header.
Message	A111 Syntax error in module header
Cause	The specification format of the control instruction specified in the module header is wrong.
User action	Correct the specification format of the control instruction.
Message	A112 Structured assembler preprocessor internal error
Cause	An internal error occured in the structured assembler.
User action	Consult NEC.
Message	A113 Can't used processor type
Cause	A model other than target model is specified by the PROCESSOR control instruction in the source file.
User action	Specify a correct model.
Message	A114 Can't used processor type -C 'model'
Cause	A model other than target model is specified by target model specification option (-C).
User action	Specify a correct target model name.
★ Message	A115 No processor mode specified
Cause	The -M option is not specified.
User action	Specify the -M option.

10.1.2 Fatal error

A fatal error message is output to the standard output unit or error file if the specification of the source program is wrong.

Unlike an abort message error, processing is continued even after a fatal error message has been output.

If an error is found, the secondary source file is deleted. If the J option is specified, however, the translation processing is not performed, but the source line is output to the secondary source file as is.

The input file name specified on the start line is output as the file name. If the file is an include file, the line number of the file is reset to 1. When the original file is restored, the line number is also restored.

(1/3)

Message	F201 Illegal #ELSE/#ENDIF
Cause	The position of the #ELSE or #ENDIF statement is wrong.
User action	Write the #ELSE or #ENDIF statement at the correct position.
Message	F202 Illegal #ENDGETI
Cause	The position of the #ENDGETI statement is wrong.
User action	Write the #ENDGETI statement at the correct position.
Message	F203 Missing #ENDIF
Cause	The #ENDIF statement is missing.
User action	Add the #ENDIF statement.
Message	F204 Missing #ENDGETI
Cause	The #ENDGETI statement is missing.
User action	Add the #ENDGETI statement.
Message	F205 Too many #DEFGETI definitions
Cause	The number of registered GETI instruction translation patterns exceeds the limit.
User action	Reduce the number of registered GETI replacement pseudoinstructions.
Message	F206 Too many GETI instructions
Cause	Too many instructions are defined between one pair of #DEFGETI and #ENDGETI.
User action	Define only one instruction between one pair of #DEFGETI and #ENDGETI.
Message	F207 Duplicate definition
Cause	The same translation pattern is defined twice.
User action	Correct the registration of #DEFGETI.
Message	F208 Symbol table overflow
Cause	The number of symbols exceeds the limit.
User action	Keep the number of symbols to within the limit.
Message	F209 Syntax error
Cause	The syntax of the specified statement is wrong.
User action	Correct the syntax.
Message	F210 Nest level error
Cause	Nesting is wrong (overflow, wrong pair of nesting, etc.).
User action	Correct the syntax.
Message	F211 Too many character in a line
Cause	The number of characters on one line exceeds the limit.
User action	Keep the number of characters on one line to within the limit (218 characters).

(2/3)

Message	F212 Too many include files
Cause	Nesting level of the include file exceeds the limit.
User action	Do not use the include pseudoinstruction in the include file.
Message	F214 Illegal BREAK
Cause	The position of the BREAK statement is wrong.
User action	Write the BREAK statement at the correct position.
Message	F215 Illegal CONTINUE
Cause	The position of the CONTINUE statement is wrong.
User action	Write the CONTINUE statement at the correct position.
Message	F216 Illegal CASE/DEFAULT/ENDS
Cause	The specification positions of the CASE, DEFAULT, and ENDS statements are wrong.
User action	Write the CASE, DEFAULT, and ENDS statements at the correct position.
Messagee	F217 Illegal ELSEIF/ELSE/ENDIF
Cause	The positions of the ELSEIF, ELSE, and ENDIF statements are wrong.
User action	Correct the ELSEIF, ELSE, and ENDIF statements.
Message	F218 Illegal NEXT
Cause	The position of the NEXT statement is wrong.
User action	Correct the NEXT statement.
Message	F219 Illegal ENDW
Cause	The position of the ENDW statement is wrong.
User action	Correct the ENDW statement.
Message	F220 Illegal UNTIL/UNTIL_BIT
Cause	The positions of the UNTIL and UNTIL_BIT statements are wrong.
User action	Correct the UNTIL and UNTIL_BIT statements.
Message	F221 Missing ENDIF
Cause	The ENDIF statement is missing.
User action	Add the ENDIF statement.
Message	F222 Missing ENDS
Cause	The ENDS statement is missing.
User action	Add the ENDS statement.
Message	F223 Missing ENDW
Cause	The ENDW statement is missing.
User action	Add the ENDW statement.
Message	F224 Missing NEXT
Cause	The NEXT statement is missing.
User action	Add the NEXT statement.
Message	F225 Missing UNTIL/UNTIL_BIT
Cause	The UNTIL and UNTIL_BIT statements are missing.
User action	Add the UNTIL and UNTIL_BIT statements.

(3/3)

Message	F226 Illegal character in a line
Cause	An illegal character is included in the source line.
User action	Delete the illegal character from the source line.
Message	F227 Illegal operand in a file
Cause	The data size of the assignment expression or compare condition expression is wrong.
User action	Specify the correct data size.
★ Message	F228 Illegal SFR access in a file
Cause	SFR that cannot be accessed is used in an assignment expression or condition expression.
User action	Check the access status of the SFR, and use the correct SFR.

10.1.3 Warning message

A warning message is not an error message but indicates that unfavorable processing is under way. This message is not counted as an error, and the processing can be continued.

Message	W301 symbol redefinition
Cause	A symbol is re-defined by the #define statement.
Program action	Regards the symbol defined later as valid.
User action	Correct the syntax if the symbol defined earlier should be valid.
Message	W302 Duplicate PROCESSOR option and control
Cause	A model specified by the model specification option (-C) on the command line differs from that specified by the processor model specification control instruction in the input source.
Program action	The model specified by the model specification option (-C) is valid and the processor model specification control instruction is ignored.
User action	Check to see if the model specified by the model specification option is correct.

10.2 Termination Processing Information

The EXIT STATUS returned by the structured assembler to the OS when the structured assembler is terminated is as follows:

EXIT (0) ... Normal termination

★ EXIT (1) ... Abnormal termination (One or more fatal error is found.)

★ EXIT (2) ... An abort error has occurred and the program execution has been stopped.

[MEMO]

APPENDIX A MAXIMUM PERFORMANCE

★ **(1) Length of one line**
254 characters (except LF and CR)

(2) Number of registered symbols
512 (except reserved word)

(3) Nesting level of control statements
31 levels

(4) Nesting level of conditional processing instructions
8 levels

(5) #defgeti pseudoinstructions
48

(6) Nesting level of #include pseudoinstructions
1 level

★ **(7) Number of operands that can be successively assigned**
33

[MEMO]

APPENDIX B LIST OF TARGET MODELS

★

Symbol		Target Device	
H	H-I	μPD750004, 750006, 750008, 75P0016, 750104, 750106, 750108, 75P0116, 750064, 750066, 750068, 75P0076, 753012, 753016, 753017, 75P3018, 753012A, 753016A, 753017A, 75P3018A, 753036, 75P3036, 753104, 753106, 753108, 75P3116, 753204, 753206, 75P3208, 75P3216, 753304[Note], 754202, 754144, 754244, 754264, 75F4264[Note], 754302, 754304, 75P4308	75XL series
		μPD75117H, 75P117H, 75217, 75218, 75P218, 75236, 75237, 75238, 75P238, 75517, 75518, 75P518, 75617A	75X series
	H-II	μPD75104, 75104A, 75106, 75108, 75108A, 75P108B, 75108F, 75P108, 75112, 75112F, 75116, 75P116, 75116F, 75116H, 75206, 75208, 75CG208, 75212A, 75216A, 75P216A, 75CG216A, 75336, 75P336, 75352A	
S		μPD75004, 75006, 75008, 75P008, 75028, 75036, 75P036, 75048, 75P048, 75064, 75066, 75068, 75P068, 75268, 75304, 75304B, 75306, 75306B, 75308, 75P308, 75308B, 75312, 75312B, 75316, 75P316, 75P316A, 75316B, 75P316B, 75328, 75P328	

Note Under development

Caution The structured assembler does not support the low-end devices of the 75X series (μPD75P402 and 75402A).

Remark H-I : High-End I
H-II : High-End II
S : Standard

[MEMO]

APPENDIX C LIST OF STATEMENTS OF STRUCTURED ASSEMBLER

★

	Assignment Statement	Operation	Skip Condition	H	S	Page
	A = #n4	A←n4	String-effect A	○	○	p.14-17
	A = @HL	A←(HL)		○	○	
	A = @HL+	A←(HL), then L←L+1	L = 0	○	○ Note 1	
	A = @HL−	A←(HL), then L←L−1	L = FH	○	○ Note 1	
	A = @rpa1	A←(rpa1)		○	○	
	A = mem	A←(mem)		○	○	
	A = reg	A←reg		○	○	
	reg1 = #n4	reg1←n4		○	○	
	reg1 = A	reg1←A		○	○	
	reg1 = #n4 (A)	reg1←n4		○	○	
	reg1 = @HL (A)	reg1←(HL)		○	○	
Note 2	reg1 = @HL+ (A)	reg1←(HL), then L←L+1		○	○ Note 1	
Note 3	reg1 = @HL− (A)	reg1←(HL), then L←L−1		○	○ Note 1	
	reg1 = @rpa1 (A)	reg1←(rpa1)		○	○	
	reg1 = mem (A)	reg1←(mem)		○	○	
	reg1 = reg1 (A)	reg1←reg1		○	○	
	XA = #n8	XA←n8		○	○	
	XA = @HL	XA←(HL)		○	○	
	XA = mem	XA←(mem)		○	○	
	XA = rp'	XA←rp'		○	○	
	XA = @PCDE	XA←$(PC_{13-8}+DE)_{ROM}$		○	○	
	XA= @PCXA	XA←$(PC_{13-8}+XA)_{ROM}$		○	○	
	XA = @BCDE	XA←$(B_{2-0}+CDE)_{ROM}$		Note 4	×	
	XA = @BCXA	XA←$(B_{2-0}+CXA)_{ROM}$		Note 4	×	
	HL = #n8	HL←n8	String-effect B	○	○	
	rp2 = #n8	rp2←n8		○	○	
	rp'1 = XA	rp'1←XA		○	○	
	rp'1 = #n8 (XA)	rp'1←n8		○	○	
	rp'1 = mem (XA)	rp'1←(mem)		○	○	
	rp'1 = @HL (XA)	rp'1←(HL)		○	○	
	rp'1 = rp' (XA)	rp'1←rp'		○	○	
	@HL = A	(HL)←A		○	○	
	@HL = XA	(HL)←XA		○	○	
	@HL = #n4 (A)	(HL)←n4		○	○	
Note 2	@HL = @HL+ (A)	(H (L+1))←(HL), then L←L+1		○	○ Note 1	
Note 3	@HL = @HL− (A)	(H (L−1))←(HL), then L←L−1		○	○ Note 1	
	@HL = @rpa1 (A)	(HL)←(rpa1)		○	○	
	@HL = mem (A)	(HL)←(mem)		○	○	

Notes 1. Translation is executed by combining MOV and INCS or DECS instructions.

2. Assignment is not correctly performed when L = 0.

3. Assignment is not correctly performed when L = FH.

4. Only described with H-II.

★	Assignment Statement	Operation	Skip Condition	H	S	Page
	@HL = reg1 (A)	(HL)←reg1		○	○	p.14-17
	@HL = #n8 (XA)	(HL)←n8		○	○	
	@HL = mem (XA)	(HL)←(mem)		○	○	
	@HL = rp' (XA)	(HL)←rp'		○	○	
	mem = A	(mem)←A		○	○	
	mem = XA	(mem)←XA	String-effect A	○	○	
	mem = #n4 (A)	(mem)←n4		○	○	
	mem = @HL (A)	(mem)←(HL)		○	○	
Note 2	mem = @HL+ (A)	(mem)←(HL), then L←L+1		○	○Note 1	
Note 3	mem = @HL− (A)	(mem)←(HL), then L←L−1		○	○Note 1	
	mem = @rpa1 (A)	(mem)←(rpa1)		○	○	
	mem = mem (A)	(mem)←(mem)		○	○	
	mem = reg1 (A)	(mem)←reg1		○	○	
	mem = #n8 (XA)	(mem)←n8		○	○	
	mem = mem (XA)	(mem)←(mem)		○	○	
	mem = @HL (XA)	(mem)←(HL)		○	○	
	mem = rp' (XA)	(mem)←rp'		○	○	
	CY = fmem.bit	CY←(fmem.bit)		○	×	
	CY = pmem.@L	CY←($pmem_{7-2}$+L_{3-2}.bit(L_{1-0}))		○	×	
	CY = @H+mem.bit	CY←(H+mem_{3-0}.bit)		○	×	
	fmem.bit = CY	(fmem.bit)←CY		○	×	
	fmem.bit = fmem.bit (CY)	(fmem.bit)←(fmem.bit)		○	×	
	fmem.bit = pmem.@L (CY)	(fmem.bit)←($pmem_{7-2}$+L_{3-2}.bit(L_{1-0}))		○	×	
	fmem.bit = @H+mem.bit (CY)	(fmem.bit)←(H+mem_{3-0}.bit)		○	×	
	pmem.@L = CY	($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))←CY		○	×	
	pmem.@L = fmem.bit (CY)	($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))←(fmem.bit)		○	×	
	pmem.@L = pmem.@L (CY)	($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))←($pmem_{7-2}$+L_{3-2}.bit(L_{1-0}))		○	×	
	pmem.@L = @H+mem.bit(CY)	($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))←(H+mem_{3-0}.bit)		○	×	
	@H+mem.bit = CY	(H+mem_{3-0}.bit)←CY		○	×	
	@H+mem.bit = fmem.bit (CY)	(H+mem_{3-0}.bit)←(fmem.bit)		○	×	
	@H+mem.bit = pmem.@L (CY)	(H+mem_{3-0}.bit)←($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))		○	×	
	@H+mem.bit = @H+mem.bit (CY)	(H+mem_{3-0}.bit)←(H+mem_{3-0}.bit)		○	×	
	A += #n4	A←A+n4	carry	○	○	p.18, 19
	A += @HL	A←A+(HL)	carry	○	○	
	A += @HL, CY	A, CY←A+(HL)+CY	—	○	○	
	XA += #n8	XA←XA+n8	carry	○	×	
	XA += rp'	XA←XA+rp'	carry	○	×	
	XA += rp', CY	XA, CY←XA+rp'+CY	—	○	×	
	rp'1 += XA	rp'1←rp'1 + XA	carry	○	×	
	rp'1 += XA, CY	rp'1, CY←rp'1+XA+CY	—	○	×	

Notes 1. Translation is executed by combining MOV and INCS or DECS instructions.

2. Assignment is not correctly performed when L = 0.

3. Assignment is not correctly performed when L = FH.

Assignment Statement	Operation	Skip Condition	H	S	Page
A– = @HL	A←A– (HL)	borrow	○	○	p.20, 21
A– = @HL, CY	A, CY←A–(HL)–CY	—	○	○	
XA– = rp'	XA←XA–rp'	borrow	○	×	
XA– = rp', CY	XA, CY←XA–rp'–CY	—	○	×	
rp'1– = XA	rp'1←rp'1–XA	borrow	○	×	
rp'1– = XA, CY	rp'1, CY←rp'1–XA–CY	—	○	×	
A &= #n4	A←A ∩ n4		○	○	p.22, 23
A &= @HL	A←A ∩ (HL)		○	○	
XA &= rp'	XA←XA ∩ rp'		○	×	
rp'1 &= XA	rp'1←rp'1 ∩ XA		○	×	
CY &= fmem.bit	CY←CY ∩ (fmem.bit)		○	○	
CY &= pmem.@L	CY←CY ∩ ($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))		○	○	
CY &= @H+mem.bit	CY←CY ∩ (H+mem_{3-0}.bit)		○	Note	
A |= #n4	A←A ∪ n4		○	○	p.24, 25
A |= @HL	A←A ∪ (HL)		○	○	
XA |= rp'	XA←XA ∪ rp'		○	×	
rp'1 |= XA	rp'1←rp'1 ∪ XA		○	×	
CY |= fmem.bit	CY←CY ∪ (fmem.bit)		○	○	
CY |= pmem.@L	CY←CY ∪ ($pmem_{7-2}$+L_{3-2}.bit(L_{1-0}))		○	○	
CY |= @H+mem.bit	CY←CY ∪ (H+mem_{3-0}.bit)		○	Note	
A ^= #n4	A←A ∀ n4		○	○	p.26, 27
A ^= @HL	A←A ∀ (HL)		○	○	
★ XA ^= rp'	XA←XA ∀ rp'		○	×	
★ rp'1 ^= XA	rp'1←rp'1 ∀ XA		○	×	
CY ^= fmem.bit	CY←CY ∀ (fmem.bit)		○	○	
CY ^= pmem.@L	CY←CY ∀ ($pmem_{7-2}$+L_{3-2}.bit (L_{1-0}))		○	○	
CY ^= @H+mem.bit	CY←CY ∀ (H+mem_{3-0}.bit)		○	Note	

Note This cannot be used with the μPD75048 when MBS = 4, 5, 6, or 7.

Increment/Decrement Statement	Operation	Skip Condition	H	S	Page
reg++	reg←reg+1	reg = 0	○	○	p.31
rp1++	rp1←rp1+1	rp1 = 00H	○	×	
@HL++	(HL)←(HL)+1	(HL) = 0	○	○	
mem++	(mem)←(mem)+1	(mem) = 0	○	○	
reg– –	reg←reg–1	reg = FH	○	○	p.32
rp'– –	rp'←rp'–1	rp = FFH	○	×	

Exchange Statement	Operation	Skip Condition	H	S	Page
A<–>@HL	A↔(HL)		○	○	p.34
★ A<–>@HL+	A↔(HL), then L←L+1	L = 0	○	○Note	
★ A<–>@HL–	A↔(HL), then L←L–1	L = F	○	○Note	
A<–>@rpa1	A↔(rpa1)		○	○	
XA<–>@HL	XA↔(HL)		○	○	
A<–>mem	A↔(mem)		○	○	
XA<–>mem	XA↔(mem)		○	○	
A<–>reg1	A↔reg1		○	○	
XA<–>rp'	XA↔rp'		○	○	

Note Translation is performed with XCH and INCS or DECS instructions combined.

[MEMO]

APPENDIX D CONTROL STATEMENT LIST

Control Statement	Description Format	Page

if statement	if (condition expression 1) [(register name)]
	statement 1
	elseif (condition expression 2) [(register name)]
	statement 2
	else
	statement 3
	endif
		p.62-64

★ switch statement[Note]	switch (symbol) [(register name)] case constant 1: statement 1 case constant 2: statement 2 : case constant N: statement N default: statement N+1 ends	p.66-70
for statement	for (assignment statement; condition expression; increment/decrement statement) [(register name)] statement next	p.71, 72
while statement	while (condition statement) [(register name)] statement endw	p.73, 74
until statement	repeat statement until (condition expression) [(register name)]	p.77, 78
break statement	break	p.81
continue statement	continue	p.82

if_bit statement	if_bit (bit condition 1)
	statement 1
	elseif_bit (bit condition 2)
	statement 2
	else
	statement 3
	endif
		p.65-67

while_bit statement	while_bit (bit condition) statement endw	p.73, 74
until_bit statement	repeat statement until_bit (bit condition)	p.79, 80
goto statement	goto label	p.84, 85
★ forever statement	• for (expression 1; forever; expression 3) • while (forever) • until (forever)	p.86

Note The symbol (α) and register (γ) that can be described in the switch statement are shown below.

★

α	γ	Operation	H	S
#n4	—	if #n4 = constant i then goto statement i	○	○
@HL[Note]	—	if @HL = constant i then goto statement i	○	○
@rpa1	—	if @rpa1 = constant i then goto statement i	○	○
mem	—	if mem = constant i then goto statement i	○	○
reg[Note]	—	if reg = constant i then goto statement i	○	○
A	reg	if A = constant i then goto statement i	○	○
A	@HL	if A = constant i then goto statement i	○	○
reg1	A	if reg1 = constant i then goto statement i	○	○
@HL	A	if @HL = constant i then goto statement i	○	○
@rpa1	A	if @rpa1 = constant i then goto statement i	○	○
#n4	reg	if #n4 = constant i then goto statement i	○	○
mem	A	if mem = constant i then goto statement i	○	○

The combinations of a condition expression and a register name that can be used in a control statement are shown below.

	Condition Statement	Register	True	False	H	S	Page
	reg == #n4	—	reg = n4	reg ≠ n4	○	○	p.39-41
	@HL == #n4	—	(HL) = n4	(HL) ≠ n4	○	○	
	@HL == #n4	A	(HL) = n4	(HL) ≠ n4	○	○	
	#n4 == #n4	A	n4 = n4	n4 ≠ n4	○	○	
	#n4 == #n4	reg1	n4 = n4	n4 ≠ n4	○	○	
★ Note 1	@HL+ == #n4	A	(HL) = n4	(HL) ≠ n4	○	○	
★ Note 2	@HL− == #n4	A	(HL) = n4	(HL) ≠ n4	○	○	
	@rpa1 == #n4	A	(rpa1) = n4	(rpa1) ≠ n4	○	○	
	mem == #n4	A	(mem) = n4	(mem) ≠ n4	○	○	
	A == #n4	@HL	A = n4	A ≠ n4	○	○	
	A == @HL	—	A = (HL)	A ≠ (HL)	○	○	
	reg1 == @HL	A	reg1 = (HL)	reg1 ≠ (HL)	○	○	
	XA == @HL	—	XA = (HL)	XA ≠ (HL)	○	×	
	rp' == @HL	XA	rp' = (HL)	rp' ≠ (HL)	○	×	
	#n4 == @HL	A	n4 = (HL)	n4 ≠ (HL)	○	○	
	@rpa1 == @HL	A	(rpa1) = (HL)	(rpa1) ≠ (HL)	○	○	
	mem == @HL	A	(mem) = (HL)	(mem) ≠ (HL)	○	○	
	mem == @HL	XA	(mem) = (HL)	(mem) ≠ (HL)	○	×	
	#n8 == @HL	XA	n8 = (HL)	n8 ≠ (HL)	○	×	
	A == reg	—	A = reg	A ≠ reg	○	○	
	reg1 == reg	A	reg1 = reg	reg1 ≠ reg	○	○	
	#n4 == reg	A	n4 = reg	n4 ≠ reg	○	○	
	@HL == reg	A	(HL) = reg	(HL) ≠ reg	○	○	
★ Note 1	@HL+ = reg	A	(HL) = reg	(HL) ≠ reg	○	○	
★ Note 2	@HL− = reg	A	(HL) = reg	(HL) ≠ reg	○	○	
	@rpa1 == reg	A	(rpa1) = reg	(rpa1) ≠ reg	○	○	
	mem == reg	A	(mem) = reg	(mem) ≠ reg	○	○	
	XA == rp'	—	XA = rp'	XA ≠ rp'	○	×	
	rp' == rp'	XA	rp' = rp'	rp' ≠ rp'	○	×	
	#n8 == rp'	XA	n8 = rp'	n8 ≠ rp'	○	×	
	mem == rp'	XA	(mem) = rp'	(mem) ≠ rp'	○	×	
	@HL == rp'	XA	(HL) = rp'	(HL) ≠ rp'	○	×	

Notes 1. The value of the L register is incremented by 1.
Comparison is not performed when L = FH.

2. The value of the L register is decremented by 1.
Comparison is not performed when L = 0H.

	Condition Statement	Register	True	False	H	S	Page
	reg! = #n4	—	reg ≠ n4	reg = n4	○	○	p.38, 39
	@HL! = #n4	—	(HL) ≠ n4	(HL) = n4	○	○	
	@HL! = #n4	A	(HL) ≠ n4	(HL) = n4	○	○	
	#n4! = #n4	A	n4 ≠ n4	n4 = n4	○	○	
	#n4! = #n4	reg1	n4 ≠ n4	n4 = n4	○	○	
★ Note 1	@HL+! = #n4	A	(HL) ≠ n4	(HL) = n4	○	○	
★ Note 2	@HL–! = #n4	A	(HL) ≠ n4	(HL) = n4	○	○	
	@rpa1! = #n4	A	(rpa1) ≠ n4	(rpa1) = n4	○	○	
	mem! = #n4	A	(mem) ≠ n4	(mem) = n4	○	○	
★	A! = #n4	@HL	A ≠ n4	A = n4	○	○	
	A! = @HL	—	A ≠ (HL)	A = (HL)	○	○	
	reg1! = @HL	A	reg1 ≠ (HL)	reg1 = (HL)	○	○	
	XA! = @HL	—	XA ≠ (HL)	XA = (HL)	○	×	
	rp'! = @HL	XA	rp' ≠ (HL)	rp' = (HL)	○	×	
	#n4! = @HL	A	n4 ≠ (HL)	n4 = (HL)	○	○	
	@rpa1! = @HL	A	(rpa1) ≠ (HL)	(rpa1) = (HL)	○	○	
	mem! = @HL	A	(mem) ≠ (HL)	(mem) = (HL)	○	○	
	mem! = @HL	XA	(mem) ≠ (HL)	(mem) = (HL)	○	×	
	#n8! = @HL	XA	n8 ≠ (HL)	n8 = (HL)	○	×	
	A! = reg	—	A ≠ reg	A = reg	○	○	
	reg1! = reg	A	reg1 ≠ reg	reg1 = reg	○	○	
	#n4! = reg	A	n4 ≠ reg	n4 = reg	○	○	
	@HL! = reg	A	(HL) ≠ reg	(HL) = reg	○	○	
★ Note 1	@HL+! = reg	A	(HL) ≠ reg	(HL) = reg	○	○	
★ Note 2	@HL–! = reg	A	(HL) ≠ reg	(HL) = reg	○	○	
	@rpa1! = reg	A	(rpa1) ≠ reg	(rpa1) = reg	○	○	
	mem! = reg	A	(mem) ≠ reg	(mem) = reg	○	○	
	XA! = rp'	—	XA ≠ rp'	XA = rp'	○	×	
	rp'! = rp'	XA	rp' = rp'	rp' = rp'	○	×	
	#n8! = rp'	XA	n8 ≠ rp'	n8 = rp'	○	×	
	mem! = rp'	XA	(mem) ≠ rp'	(mem) = rp'	○	×	
	@HL! = rp'	XA	(HL) ≠ rp'	(HL) = rp'	○	×	
	A < @HL	—	A < (HL)	A >= (HL)	○	○	p.40, 41
	reg1 < @HL	A	reg1 < (HL)	reg1 >= (HL)	○	○	
	#n4 < @HL	A	n4 < (HL)	n4 >= (HL)	○	○	
	@rpa1 < @HL	A	(rpa1) < (HL)	(rpa1) >= (HL)	○	○	
	mem < @HL	A	(mem) < (HL)	(mem) >= (HL)	○	○	
	XA < rp'	—	XA < rp'	XA >= rp'	○	×	
	rp' < rp'	XA	rp' < rp'	rp' >= rp'	○	×	
	#n8 < rp'	XA	n8 < rp'	n8 >= rp'	○	×	
	@HL < rp'	XA	(HL) < rp'	(HL) >= rp'	○	×	
	mem < rp'	XA	(mem) < rp'	(mem) >= rp'	○	×	
	rp'1 < XA	—	rp'1 < XA	rp'1 >= XA	○	×	
★	#n8 < XA	rp'1	n8 < XA	n8 >= XA	○	×	

Notes 1. The value of the L register is incremented by 1.
Comparison is not performed when L = FH.

2. The value of the L register is decremented by 1.
Comparison is not performed when L = 0H.

Condition Statement	Register	True	False	H	S	Page
@HL > A	—	(HL) > A	(HL) <= A	○	○	p.42, 43
@HL > reg1	A	(HL) > reg1	(HL) <= reg1	○	○	
@HL > #n4	A	(HL) > n4	(HL) <= n4	○	○	
@HL > @rpa1	A	(HL) > (rpa1)	(HL) <= (rpa1)	○	○	
@HL > mem	A	(HL) > (mem)	(HL) <= (mem)	○	○	
rp' > XA	—	rp' > XA	rp' <= XA	○	×	
rp' > rp'	XA	rp' > rp'	rp' <= rp'	○	×	
rp' > #n8	XA	rp' > n8	rp' <= n8	○	×	
rp' > @HL	XA	rp' > (HL)	rp' <= (HL)	○	×	
rp' > mem	XA	rp' > (mem)	rp' <= (mem)	○	×	
XA > rp'1	—	XA > rp'1	XA <= rp'1	○	×	
★ XA > #n8	rp'	XA > n8	XA <= n8	○	×	
A >= @HL	—	A >= (HL)	A < (HL)	○	○	p.44, 45
reg1 >= @HL	A	reg1 >= (HL)	reg1 < (HL)	○	○	
#n4 >= @HL	A	n4 >= (HL)	n4 < (HL)	○	○	
@rpa1 >= @HL	A	(rpa1) >= (HL)	(rpa1) < (HL)	○	○	
mem >= @HL	A	(mem) >= (HL)	(mem) < (HL)	○	○	
XA >= rp'	—	XA >= rp'	XA <rp'	○	×	
rp' >= rp'	XA	rp' >= rp'	rp' < rp'	○	×	
#n8 >= rp'	XA	n8 >= rp'	n8 < rp'	○	×	
@HL >= rp'	XA	(HL) >= rp'	(HL) < rp'	○	×	
mem >= rp'	XA	(mem) >= rp'	(mem) < rp'	○	×	
rp'1 >= XA	—	rp'1 >= XA	rp'1 < XA	○	×	
★ #n8 >= XA	rp'1	n8 >= XA	n8 < XA	○	×	
@HL <= A	—	(HL) <= A	(HL) > A	○	○	p.46, 47
@H <= reg1	A	(HL) <= reg1	(HL) > reg1	○	○	
@HL <= #n4	A	(HL) <= n4	(HL) > n4	○	○	
@HL <= @rpa1	A	(HL) <= (rpa1)	(HL) > (rpa1)	○	○	
@HL <= mem	A	(HL) <= (mem)	(HL) > (mem)	○	○	
rp' <= XA	—	rp' <= XA	rp' > XA	○	×	
rp' <= rp'	XA	rp' <= rp'	rp' > rp'	○	×	
rp' <= #n8	XA	rp' <= n8	rp' > n8	○	×	
rp' <= @HL	XA	rp' <= (HL)	rp' > (HL)	○	×	
rp' <= mem	XA	rp' <= (mem)	rp' > (mem)	○	×	
XA <= rp'1	—	XA <= rp'1	XA >rp'1	○	×	
★ XA <= #n8	rp'1	XA <= n8	XA > n8	○	×	

[MEMO]

APPENDIX E PSEUDOINSTRUCTION LIST

Pseudoinstruction	Format	Page
#define pseudoinstruction	#define symbol character string	p.88, 89
#ifdef pseudoinstruction	#ifdef symbol text 1 #else text 2 #endif	p.90
#include pseudoinstruction	#include "file name"	p.91
#defgeti pseudoinstruction	#defgeti label of GETI table instruction pattern : #endgeti	p.92, 93

[MEMO]

★ APPENDIX F CONTROL INSTRUCTION LIST

Control Instruction	Format	Page
$PROCESSOR control instruction	$ [△] PROCESSOR [△] = [△] model name [△] [; comment] $ [△] PC [△] = [△] model name [△] [; comment]	p.97
$MODE control instruction	$ [△] MODE [△] = [△] constant [△] [; comment] $ [△] MD [△] = [△] constant [△] [; comment]	p.100

[MEMO]

APPENDIX G OPTION LIST

Option	Option Name	Format	Default Assumption	Page
C	Model specification	-C model name	Must not be omitted.	p.111, 112
D	Symbol definition specification	-D symbol [= numeric value]	symbol = 1	p.113
WT	Number of tabs specification	-WT numeric value 1, numeric value 2, numeric value 3	numeric value 1 = 2, numeric value 2 = 3, numeric value 3 = 4	p.114
I	Include file path specification	-I [drive number:] directory	It is assumed that current drive and current directory are specified.	p.115
O	Secondary source file specification	-O [drive number:] [directory] output file name	Creates file that replaces file type of input file with ".ASM" in current directory.	p.116
E	Error list file specification	-E [drive number:] [directory] output file name	Creates file that replaces file type of input file with ".EST" in current directory.	p.117
F	Parameter file specification	-F[drive number:] [directory] output file name	If file type of input file is omitted, ".PST" is assumed.	p.118
J	Secondary source file forced output specification	-J	—	p.119
★ M	Mode specification	-M mode name	—	p.120
★ S, NS	Symbol name length specification	-S, -NS	31 characters	p.121
★ GS, NGS	Debug information output specification	-GS, -NGS	Output	p.122
★ Y	Device file search path	-Y [drive number:] directory	—	p.123
–	Help specification	– –	—	p.124

–: None

[MEMO]

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Technical Documentation Dept.
Fax: +49-211-6503-274

South America
NEC do Brasil S.A.
Fax: +55-11-889-1689

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Corporation
Semiconductor Solution Engineering Division
Technical Information Support Dept.
Fax: 044-548-7900

I would like to report the following error/make the following suggestion:

Document title: _____

Document number: _____ Page number: _____

_____

_____

_____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❑	❑	❑	❑
Technical Accuracy	❑	❑	❑	❑
Organization	❑	❑	❑	❑