To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

User's Manual

RA78K0S Series

Assembler Package

Structured Assembler Language

ST78K0S V1.00 or Later

Document No. U11623EJ1V0UMJ1 (1st edition)
Date Published January 2001 N CP(K)

[MEMO]

- **The information in this document is current as of August, 1996. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC's data sheets or data books, etc., for the most up-to-date specifications of NEC semiconductor products. Not all products and/or types are available in every country. Please check with an NEC sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without prior written consent of NEC. NEC assumes no responsibility for any errors that may appear in this document.
- NEC does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC semiconductor products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of customer's equipment shall be done under the full responsibility of customer. NEC assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC endeavours to enhance the quality, reliability and safety of NEC semiconductor products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC semiconductor products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment, and anti-failure features.
- NEC semiconductor products are classified into the following three quality grades:
  "Standard", "Special" and "Specific". The "Specific" quality grade applies only to semiconductor products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of a semiconductor product depend on its quality grade, as indicated below. Customers must check the quality grade of each semiconductor product before using it in a particular application.

  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots

  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support)

  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

  The quality grade of NEC semiconductor products is "Standard" unless otherwise expressly specified in NEC's data sheets or data books, etc. If customers wish to use NEC semiconductor products in applications not intended by NEC, they must contact an NEC sales representative in advance to determine NEC's willingness to support a given application.

  (Note)

  (1) "NEC" as used in this statement means NEC Corporation and also includes its majority-owned subsidiaries.

  (2) "NEC semiconductor products" means any semiconductor product developed or manufactured by or for NEC (as defined above).

M8E 00.4

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: 408-588-6000
800-366-9782
Fax: 408-588-6130
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.l.
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.
Madrid Office
Madrid, Spain
Tel: 91-504-2787
Fax: 91-504-2860

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
Tel: 65-253-8311
Fax: 65-250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC do Brasil S.A.
Electron Devices Division
Guarulhos-SP Brasil
Tel: 55-11-6462-6810
Fax: 55-11-6462-6829

PREFACE

This manual has been written to help users obtain an accurate understanding of the coding method used for the structured assembler preprocessor (hereafter referred to as the “structured assembler”) that is included in the “RA78K0S Assembler Package”, i.e., the assembler package for compact, general-purpose microcontrollers in the 78K/0 Series.

This manual does not explain methods for using programs other than the structured assembler nor does it describe structured assembler operation methods.

Therefore, when writing programs, please refer to the **“ASSEMBLER PACKAGE USER’S MANUAL” (ASSEMBLY LANGUAGE AND OPERATION)**.

Target readers

This manual was written for readers who understand the functions of compact, general-purpose microcontrollers in the 78K/0 Series.

Readers requiring a description of the functions of compact, general-purpose microcontrollers in the 78K/0 Series should refer to the target chip’s User’s Manual.

Target chips

This assembler can be used for all chips supported by the RA78K0S Assembler Package.

Composition

The composition of this manual is described below.

- CHAPTER 1 GENERAL
  This chapter describes the functions (the role, etc.) of the structured assembler in software development for microcontrollers.
- CHAPTER 2 SOURCE PROGRAM CODING METHODS
  This chapter describes methods for source program configuration, coding syntax, and other principal rules and conventions concerning the coding of source programs.
- CHAPTER 3 CONTROL STATEMENTS
  Control statements are used to describe the “if~else~endif” indicators of the program structure.
  This chapter describes control statement functions and coding methods.
- CHAPTER 4 EXPRESSIONS
  Assignments and arithmetic operations are entered as expressions.
  This chapter describes expression functions and coding methods.
- CHAPTER 5 DIRECTIVES
  This chapter presents use examples in describing how to write and use structured assembler directives.
- CHAPTER 6 CONTROL INSTRUCTIONS
  This chapter presents use examples in describing how to write and use structured assembler control instructions.
- APPENDIX A SYNTAX LISTS
  This appendix presents a structured assembler syntax list.
- APPENDIX B LISTS OF GENERATED INSTRUCTIONS
  This appendix presents a list of instructions generated by the structured assembler.
- APPENDIX C MAXIMUM PERFORMANCE
  This appendix describes the maximum performance features of the structured assembler.

Use

Readers who are using a structured assembler for the first time should read this manual starting from **"CHAPTER 1 GENERAL"**.

Readers who already have a general knowledge of structured assemblers may skip Chapter 1.

However, all readers should read section **"1.3 Before Starting Program Development"**.

Legend

The meanings of common symbols in this manual are described below.

...	:	Same format or pattern is repeated
[ ]	:	Characters enclosed in these brackets can be omitted.
⌈ ⌋	:	Characters enclosed in these brackets are a character string.
''	:	Characters between single quotation marks are a character string.
""	:	Quotation marks indicate a location of reference.
Δ	:	Indicates one or more white-space characters or tabs.
Boldface	:	Characters in boldface are as shown.
__	:	Underlining is used to indicate input character strings.
:	:	Indicates that program description is omitted
( )	:	Characters between parentheses are a character string.
CR	:	Carriage return
LF	:	Line feed
/	:	Delimiter
α	:	is entered as a mnemonic operand, such as a register name
β	:	is entered as a mnemonic operand, such as a register name
γ	:	is entered as a mnemonic operand, such as a register name
δ	:	is entered as a mnemonic operand, such as a register name

CONTENTS

List of Figures

List of Tables (1/2)

List of Tables (2/2)

[MEMO]

CHAPTER 1 GENERAL

This chapter describes the functions (the role, etc.) of the ST78K0S in software development for microcontrollers.

1.1 Overview of Structured Assembler

The RA78K0S structured assembler preprocessor is a program in the “RA78K0S Assembler Package” that is used for software development of compact, general-purpose microcontrollers in the 78K/0 Series.

In this manual, the structured assembler preprocessor is abbreviated as the “structured assembler” or the “ST78K0S (structured assembler)”.

A structured assembler converts structured assembly statements such as “if~else~endif” and “for~next” into assembly language source program files. Control statements are used to enter “if~else~endif” and “for~next” descriptions.

As such, a structured assembler offers the following three advantages.

<1> Programs are easy to write

- Each program structure can be written as is, which facilitates the development process from design to coding.
- There is no need to consider label names for branching.
- Transfer instructions that contain large amounts of code can be entered as assignment statements.

<2> Programs are easy to read.

- Program structure is easy to understand.
- Operations and transfers between memory registers can be entered in a single statement.
- Other programmers’ programs are easy to read.
- Program maintenance (revision) is easy.

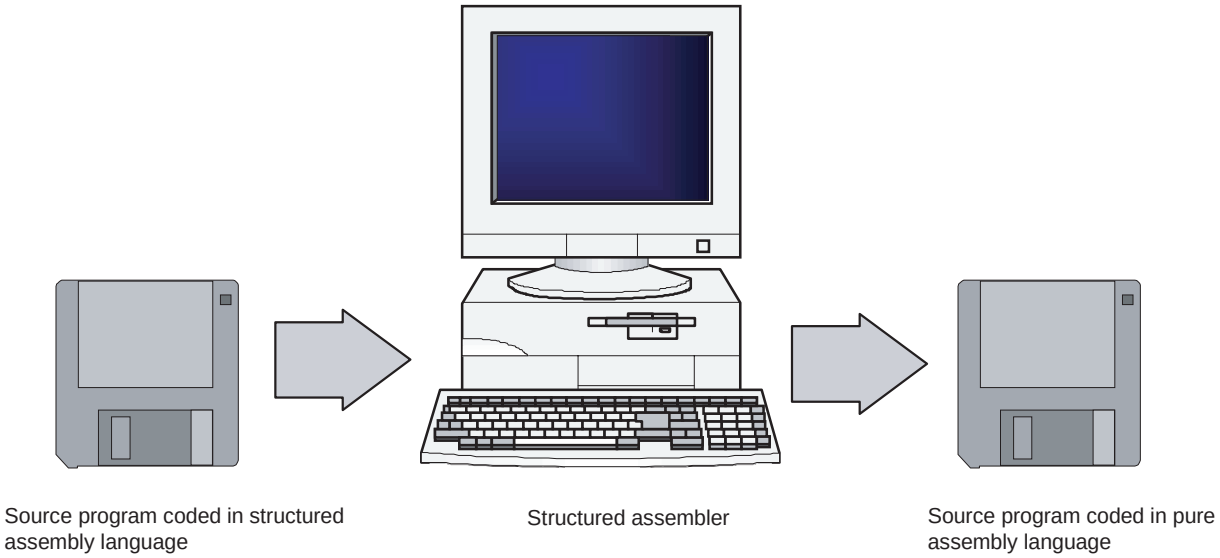
<3> Facilitates desktop debugging

- Coding can be done on a one-to-one correspondence with the detail design, thus facilitating desktop debugging.

1.2 Overview of Functions

The structured assembler analyzes various control statements, expressions, and directives within a structured assembler source program that are coded according to a specific language specification and outputs an assembler source program that serves as an input source file for the assembler.

Figure 1-1. Structured Assembler Function



Structured statements can be output as comments and converted assembler instructions and ordinary assembly language can all be output as secondary source files.

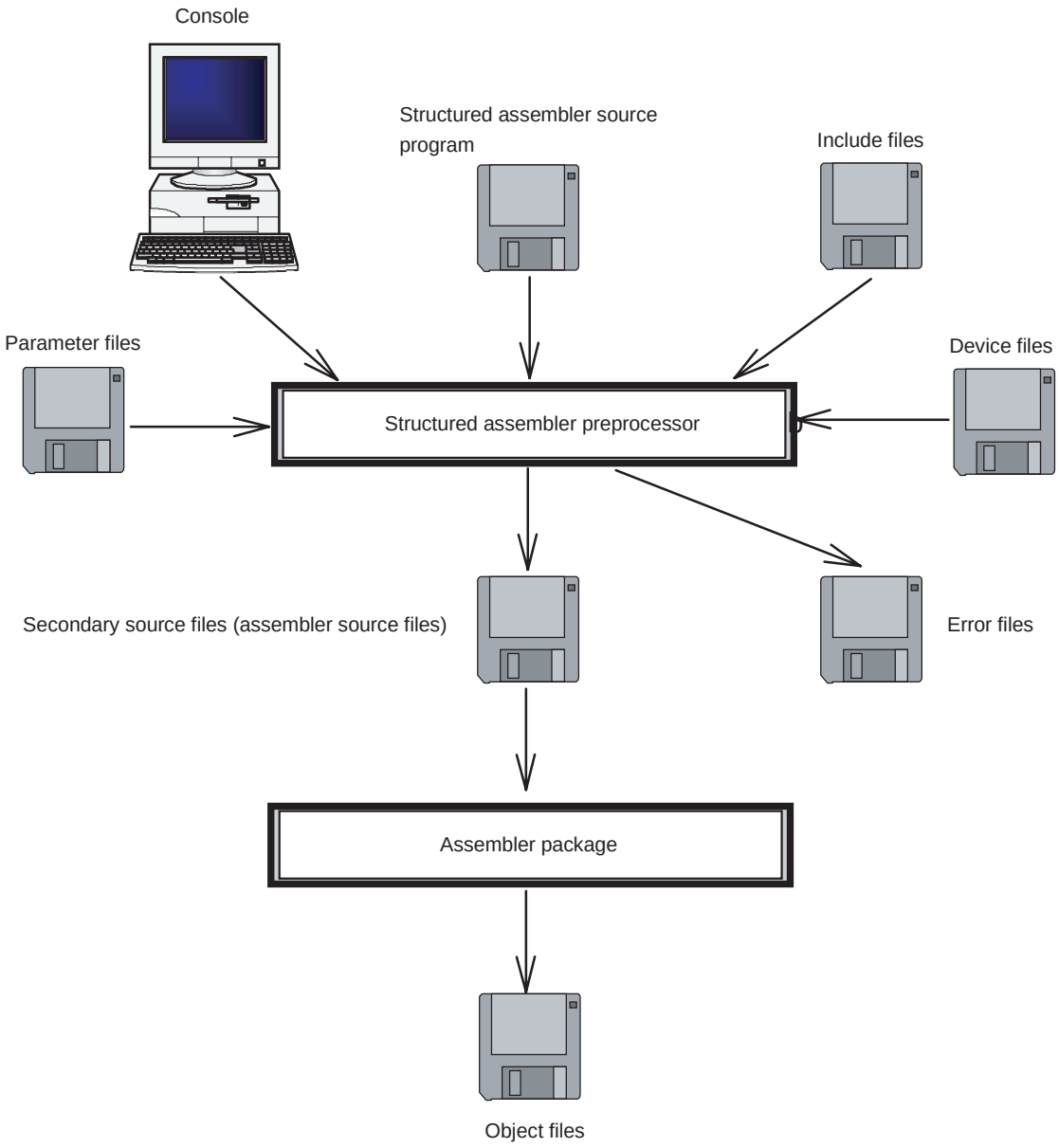
Error messages are output when errors occur.

The main functions of the structured assembler are listed below.

<1> Program coding is facilitated by an abundance of C-like control statements.
<2> C-like assignment statements and assignment operators can be used in coding.
<3> Control structures and assignment statements can be coded for bit processing.
<4> It includes C-like symbol definition directives, conditional processing functions, and include directives.
<5> Since it is the preprocessor that outputs assembler source programs, code optimization can be performed following conversion by the structured assembler.
<6> A directive is provided for converting to CALLT instructions, so that routines can be registered to a CALLT table following development of a program.
<7> Easy-to-read assembly lists can be created by changing the assembler source program output position.

Figure 1-2 shows a flowchart of program development.

Figure 1-2. Program Development Flowchart



Caution: Device files are sold separately.

1.3 Before Starting Program Development

The maximum performance features of the structured assembler are listed below. Be sure to refer to these values before writing programs.

1.3.1 Maximum performance

The structured assembler's maximum performance values are listed below.

Table 1-1. Maximum Performance of Structured Assembler

Item	Maximum value
Line length (not including LF or CR)	218 characters
Number of symbols registered in #define directive (excluding reserved words)	512 symbols
Nesting levels in control statement	31 levels
Nesting levels in #ifdef directive	8 levels
#defcallt directives	32
Nesting of #include directives	Not supported
Number of redefinitions by #define directive	31 times
Number of operands assigned in a series	33 (Note 1)
Logical operator operands	17 (Note 2)
Number of symbols defined by -D option	30

Notes 1. The maximum value is expressed as follows.
S1=S2= ··· S32=S33
Up to 33 symbols, including 32 equal signs (=), can be inserted.

2. The maximum value is expressed as follows.
expression 1&&expression 2&& ... &&expression 16&&expression 17
Up to 17 expressions and 16 "&&" (or "||") signs can be inserted.

1.3.2 Caution points

(1) Word symbols and byte symbols

The structured assembler uses the last character in each user symbol to determine whether the symbol is a word symbol or a byte symbol. The default character for word symbols is “P”, and it can be changed via the -SC option.

For details of the -SC option, see the **“RA78K0S Series Assembler Package Operation”**.

Example 1

Structured assembler

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

Assembler

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

Example 2 Start command for structured assembler

A>ST78K3 INPUT.A -SC@

“@” is used as the character indicating a word symbol.

Input file specification

Structured assembler command name

Structured assembler

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

Assembler

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

(2) Definition of label (symbol indicating address via assembler)

When defining labels, be sure to enter the label definition on a separate line from the structured assembler statement.

Example

Good example

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

Bad example

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

1.3.3 Environment variables

LANG78K specifies the type of kanji code used for entering comments.

(1) Coding format

SET∆LANG78K = kanji code

Kanji codes

SJIS : Shift JIS code

EUC : EUC code

NONE : No kanji code processing

(2) Functions

- If no environment variable has been set, the kanji code specification is set according to the OS, as follows.

  MS-DOS : SJIS

  PC DOS : NONE

  SunOS : EUC

  HP-UX : SJIS

  NEWS-OS : SJIS

- The priority of kanji code specifications is as follows.

  <1> Specification by -ZS, -ZE, or -ZN option

  <2> Specification by kanji code specification control instruction ($KANJICODE)

  <3> Specification by LANG78K environment variable

  <4> Default specification based on OS

CHAPTER 2 SOURCE PROGRAM CODING METHODS

This chapter describes coding methods and formats for source programs.

2.1 Basic Configuration of Source Programs

Source programs consist of structured assembly language and (pure) assembly language.

For further description of assembly language, see the "**RA78K0S Series Assembler Package Language**".

Each line (between two LFs) can contain up to 218 characters.

The types of coding used in structured assembly language are listed below in **Table 2-1. Structured Assembly Language Coding**.

Table 2-1. Structured Assembly Language Coding

Type			Coding
Structured assembly statement	Control statement	Conditional branch	if~elseif~else~endif if_bit~elseif_bit~else~endif switch~case~default~ends
		Conditional loop	for~next while~endw while_bit~endw repeat~until repeat~until_bit
		Other	break, continue, goto
	Expression	Assignment statement Count statement Exchange statement	Assignment (=), assignment plus operation (+=, etc.), shift (rotate) assignment (>>=, etc.) Increment (++), decrement (– –) Exchange (<->)
Conditional expression		Comparison expression Test bit expression Logical operation	==, !=, <, >, >=, <= bit address, !bit address Logical AND (&&), Logical OR (||)

Conditional expressions are entered as control statement conditions.

For details, see "**3.5 Control Statement Functions**".

(1) Control statements

Control statements include “if” and “switch~case” statements that represent conditional branches, “for~next”, “while”, and “repeat~until” statements that represent conditional loops, and “break”, “continue”, and “goto” statements that represent loop exit processing. For details, see **“CHAPTER 3 CONTROL STATEMENTS”**.

(2) Expressions

Expressions include assignment statements, count statements (increment and decrement), and exchange statements. For details, see **“CHAPTER 4 EXPRESSIONS”**.

2.2 Source Program Elements

(1) Character set

Letters, numerals, and special characters can be used in source programs.

Table 2-2. Alphanumeric Characters

Numerals		0 1 2 3 4 5 6 7 8 9
Letters	Upper case	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
	Lower case	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

In the ST78K0S, only the first character in control statements are case-sensitive. Any lower case letters that appear after the first character are converted to upper case letters. However, secondary source files are output using the case specifications in which they were entered.

Table 2-3. Special Characters

Character	Name	Use
?	Question mark	Character used as letter
@	Unit price symbol	Character used as letter
_	Underlining	Character used as letter
	White space	Delimiter symbol for phrases
HT	Horizontal tab	Character used as white space
,	Comma	Delimiter symbol for operands
.	Period	Bit position symbol for bit symbols
“	Double quotation mark	Specification character for #INCLUDE directive's disk-type file names
‘	Single quotation mark	Symbol used to mark start and end of character constant
+	Plus symbol	Positive sign or increment operation
–	Minus symbol	Negative sign or decrement operation
&	Ampersand	Logical AND operator
|	Separator symbol	Logical OR operator
ˆ	Upward arrow symbol	Exclusive OR operator
(	Left parenthesis	Change in operation sequence or expression in control statement
)	Right parenthesis	
=	Equal symbol	Assignment operator, comparison operator
:	Colon	Delimiter symbol for labels
;	Semicolon	Comment start symbol or delimiter symbol in control statement expressions
#	Number symbol or sharp symbol (in musical notation)	First character in structured assembler directive or immediate display symbol
$	Dollar sign	Location or counter value Display symbol in control instruction
!	Exclamation point	Direct addressing specification symbol, negation display symbol
<	Not equal (less than) symbol	Comparison operator
>	Not equal (more than) symbol	
\	Back slash	Directory specification symbol
[	Left bracket	Indirect address specification symbol
]	Right bracket	
LF	Line feed	End of line symbol

An error will occur if any of the following invalid characters are entered.

Table 2-4. Invalid Characters

	ASCII code
Illegal characters	00H to 08H, 0BH, 0CH, 0EH to 1FH, 7FH
Unrecognized special characters	% (25H), ‘ (60H), {(7BH),} (7DH), ¯(7EH)
Other characters	80H~0FFH

When an illegal character is entered, an error occurs and each illegal character is replaced by a period (.) when a secondary file is output.
However, invalid characters can be used in comments.

(2) Identifiers

Identifiers are names that are attached to numerical data, addresses, etc.
Identifiers are used to make the contents of source programs easier to identify.
Use #define statements to define details of identifiers (see also “**5.2 Directive Functions**”).

(3) Symbols

The last character in the symbol name determines whether the structured assembler generates a byte access instruction or a word access instruction. The default setting is P (pair), which can be changed via the -SC option.
All character strings other than reserved word symbols can be handled as user symbols. All alphanumeric characters and all other characters that can be established as English alphabet characters can be used as user symbols.

(4) Constants

Structured assembly language does not include any constants. However, assembly language constants can be output as is to secondary files (for details of assembly language constants, refer to the **“RA78K0S ASSEMBLER PACKAGE USER’S MANUAL ASSEMBLY LANGUAGE”**

(5) Expressions

Expressions are constants, special characters, and symbols that are combined using operators (for details of assembly language expressions, see the **"ASSEMBLER PACKAGE USER'S MANUAL ASSEMBLY LANGUAGE"**.

Be sure to enclose in parentheses any symbols that are separated by white spaces within an assembly language expression.

Examples

<1> Coding method for assembler

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

<2> Coding method for structured assembler source program

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

2.3 Reserved Words

The following table lists reserved words in structured assembly language.
For information on instructions and sfr symbols, see the target device's User's Manual.

Table 2-5. Reserved Word Symbols (1/2)

	Reserved word
Control statements	IF, IF_BIT, ELSEIF, ELSEIF_BIT, ELSE, ENDIF
	SWITCH, CASE, DEFAULT, ENDS
	FOR, NEXT
	WHILE, WHILE_BIT, ENDW
	REPEAT, UNTIL, UNTIL_BIT
	BREAK, CONTINUE, GOTO
Directives	DFINE
	IFDEF, ELSE, ENDIF
	INCLUDE
	DEFCALLT, ENDCALLT
Operators	++, – –
	=, +=, –=, &=, |=, ^=, <<=, >>=, <–>
	==, !=, <, >=, >, <=, FOREVER
Assembler operators	MOD, NOT
	AND, OR, XOR
	EQ, NE, GT, GE, LT, LE
	SHL, SHR
	HIGH, LOW
	DATAPOS, BITPOS, MASK

Table 2-5. Reserved Word Symbols (2/2)

	Reserved word
Assembler control instructions	PROCESSOR, PC
	DEBAG, NODEBAG, DEBAGA, NODEBAGA, , DG, NODG
	XREF, XR, NOXREF, NOXR
	TITLE, TI
	SYMLEN, NOSYMLEN
	CAP, NOCAP
	SYMLIST, NOSYMLIST
	FORMFEED, NOFORMFEED
	WIDTH, LENGTH
	TAB
	KANJICODE
	IC
	EJECT, EJ
	LIST, LI, NOLIST, NOLI
	GEN, NOGEN
	COND, NOCOND
	SUBTITLE, ST
	SET, RESET
	_IF, _ELSEIF,
Registers	CY, Z
	A
	R0, R1, R2, R3, R4, R5, R6, R7, X, B, C, D, E, H, L
	PSW
	AX
	RP0, RP1, RP2, RP3, BC, DE, HL
	SP
Other	DGS, DGL, TOL_INF, SJIS, EUC, NONE

2.4 Label Generation Rules

When using control statements in assembler instructions, the structured assembler generates labels for branch instructions.

Labels generated by the structured assembler have the format “?Ldddd”.

The “dddd” represents a decimal value of 1 or more, output with suppression of zeros and left alignment. Therefore, do not enter any labels using this “?Ldddd” format.

2.5 Size Specification

Size specifications can be made to change the data size of symbols entered in the left or right sides of an assignment expression or a conditional expression or case symbols in switch statements.

[Coding format]

(Δsize_specification_characterΔ)

[Function]

<1> If the size character is either “B” or “b”, the data size is changed to bytes.

[Description]

<1> An error will occur if the size specification character is incorrect.

<2> An error will occur if a size specification is entered in an assignment expression or a conditional expression which does not support size specifications.

<3> If a size specification is made to a register, coding can only be done using the same specification. The data size cannot be changed. If the data size is different, an error will occur.

<4> When specifying a user symbol, be sure to change the data size to the specified data size.

<5> If a size specification has been entered for a direct access specification symbol or an indirect access specification symbol or for immediate data, the size specification will be ignored and the data size will not be changed.

<6> Word access cannot be specified in size specifications.

2.6 Data Sizes

The structured assembler checks the data size of symbols. This is because the symbols differ according to the instruction being generated. However, the structured assembler allows the assembler to determine whether or not the symbol definitions and constants are entered correctly.

The data sizes checked by the structured assembler are listed below.

Table 2-6. Data Sizes

a	CY
b	Bit symbols (except [HL].β) This structured assembler recognizes bit sfrs and symbols entered using the format "$\alpha.\ \beta$" as bit symbols. Items that can be entered as "α" include byte user symbols, word user symbols, byte-specified user symbols, sfrs, constants, A, and PSW. Items that can be entered as "β" include byte user symbols, word user symbols, and constants.
c	[HL]. β Items that can be entered as "β" include byte user symbols, word user symbols, and constants.
d	Byte user symbol
e	byte-specified user symbols, sfrs that overlaps saddrp
f	A
g	Byte registers (except A, R0, R1)
h	R0
i	R1
j	sfr
k	PSW
l	Word user symbol
m	sfrp that overlaps saddrp
n	AX
o	Word register (except AX, RP0)
p	RP0
q	sfrp
r	SP
s	Direct access specification symbols These are symbols that are specified using the format "!addr". Byte user symbols, word user symbols, constants, and "$" can be entered as "addr".
t	Indirect access specification symbols These are symbols that are specified using the format "[HL]" or "[HL+byte]". Byte user symbols, constants, and "$" can be entered as "byte".
u	Special indirect access specification symbols These are symbols that are specified using the format "[DE]".
v	Immediate data These are symbols that are specified using the format "#date". Byte user symbols, word user symbols, constants, and "$" can be entered as "date".

2.7 Comments

Any character string that appears after a semicolon (;) until the next line feed (LF) is regarded as a comment statement, which is not processed but is simply output to a secondary file. Comment statements can be entered at any position in a line of code.

However, since semicolons are used between parentheses as expression delimiters in the “for~next” syntax, the two semicolons that are entered between parentheses are not regarded as the start of a comment statement.

All of the characters listed under **“2.2 (1) Character set”** can be used in comments.

Processing of illegal characters does not occur when the illegal characters are included in a comment or comment statement.

2.8 Tool Information

The structured assembler outputs tool information.

If an input source file contains tool information that has been output by the structured assembler, the “$” character at the start of the information is replaced with “;”.

The output position is the end of the module header. The only types of statements that can be entered in module headers are assembler control instructions, comment statements, and line feeds.

[Output format]

$TOL_INF 2FH, second parameter, third parameter, 0FFFFH

2FH indicates that it is tool information output by the structured assembler preprocessor.

The second parameter indicates the version number of this preprocessor.

The version number is output either as a hexadecimal value or, if the value is not converted, as the decimal number image that was shown at startup.

(Example)

Version number 1.00 → 100H

The third parameter is used to indicate this preprocessor's error messages.

0H	Normal end
1H	Fatal error, exited
2H	Warning, exited
3H	Fatal error and warning, exited

0FFFFH indicates language-related information. This is a fixed value for this preprocessor.

2.9 Output Results of Input Source Files by Structured Assembler

Input source files are output as follows by the structured assembler.

Table 2-7. Output by Structured Assembler

Input source program file	Secondary source program file
Structured assembler control statements Structured assembler expression statements	Output as comments
Structured assembler directives	Not output
#INCLUDE	Outputs include contents
Source alias set by #IFDEF	Not output
Comments	Output as comments
Other lines	Output as is

CHAPTER 3 CONTROL STATEMENTS

This chapter presents examples in describing control statement functions.

3.1 Overview of Control Statements

Control statements are used to structurally code the flow of program control.
Control statements include the followings.

- Conditional branch (IF~THEN~ELSE)
  (1) if~elseif~else~endif
  (2) if_bit~elseif_bit~else~endif
  (3) switch~case~default~ends

- Conditional loop (DO~WHILE)
  (4) for~next (loop increment)
  (5) while~endw (loop condition judgment before processing)
  (6) while_bit~endw (loop condition judgment before processing)
  (7) repeat~until (loop condition judgment after processing)
  (8) repeat~until_bit (loop condition judgment after processing)
  (9) break (Loop block break)
  (10) continue (Loop block loop)
  (11) goto (Exit for exception handling)

3.2 Control Statement Characters

The instruction generated by a control statement differs fundamentally depending on whether upper case or lower case letters are used in the control statement. For example, the different statement sizes between “if~endif” and “IF~ENDIF” can preclude direct branching via the conditional branch instruction generated by processing of the condition expression.

However, ensuring that the statement will always be branched correctly has the disadvantage of reducing the program’s efficiency as an object.

As a solution to this problem, the user is able to set upper or lower case in order to improve the object efficiency rate. If there is no need to improve the object efficiency rate, the user can omit changing the character size as long as coding uses upper case letters.

Since control statements generate conditional branch instructions, be sure to specify whether or not the relative address is within 128 bytes.

In control statements, “if” and “elseif” are reserved words. The structured assembler determines whether the first character in a control statement reserved word is an upper case or lower case letter.

IF, If ··· First letter is upper case, so coding is determined as upper case.
if, iF ··· First letter is lower case, so coding is determined as lower case.

If entered in upper case ··· branches using a combination of conditional branch instruction and BR directive.
If entered in lower case ··· branches directly using a conditional directive.

Paired control statements (such as “if, else,endif”) can have mixed upper case and lower case letters. In other words, it is possible to enter one as “IF~else~ENDIF”.

3.3 Nesting

Control statements can be nested. Generally, up to 31 nesting levels are allowed. However, control statements cannot be intersected.

(Example of incorrect coding)

```
while (A < B)
  if (A == #4)
    break;
endw
  endif
```

Error occurs due to intersecting.

(Example of correct coding)

```
while (A < B)
  if (A == #4)
    break;
  endif
endw
```

IF statement is correctly nested within WHILE statement.

3.4 Register Specification

[Coding format]

```
([Δ] [=] [Δ] register name [Δ])
```

[Function]

<1> If a register is specified immediately after a comparison expression

After the instruction to transfer the left side to the specified register, a comparison expression is generated to compare the specified register with the right side.

(Example)

```
            CMP     SYM1,#5 ;if(SYM1!=#5 && SYM2>=#0 && SYM3<#80H(A))
            BZ      $?L1
            CMP     SYM2,#0
            BC      $?L1
            MOV     A,SYM3
            CMP     A,#80H
            BNC     $?L1
?L1:                        ;endif
```

<2> If a register is specified after a control statement

During the generated of each comparison expression, after the instruction for transferring the left side to the specified register is generated, a comparison expression is generated to compare the specified register with the right side.

(Example)

```
            MOV     A,R4    ;if(R4!=#5 && R2>=#0 && R3<#80H )(A)
            CMP     A,#5
            BZ      $?L2
            MOV     A,R2
            CMP     A,#0
            BC      $?L2
            MOV     A,R3
            CMP     A,#80H
            BNC     $?L2
?L2:                        ;endif
```

<3> If both (a) and (b) are specified

The register specification that immediately follows a comparison expression takes priority. After the instruction for transferring the left side to the specified register is generated, a comparison expression is generated to compare the specified register with the right side.

As for an expression in which there is no register specification immediately after a comparison expression, after the instruction for transferring the left side to the specified register is generated according to the register specification following the control statement, a comparison expression is generated to compare the specified register with the right side.

(Example)

```
            MOV     A,DATA1 ;if(DATA1!=#5 && DATA2>=#0(A) && DATA3<#80H )(A)
            CMP     A,#5
            BZ      $?L3
            MOV     A,DATA2
            CMP     A,#0
            BC      $?L3
            MOV     A,DATA3
            CMP     A,#80H
            BNC     $?L3
?L3:                        ;endif
```

[Description]

<1> Register specifications can be used in if statements, elseif statements, switch statements, for statements, while statements, and until statements. However, if the conditional expression is a bit expression, any register specified in the control statement is ignored.

<2> For a list of register names, see **Table 2-5. Reserved Word Symbols**. sfr specifications can also be entered.

<3> The processing for an assignment statement within a for statement is the same as for comparison expressions.

3.5 Control Statement Functions

The following pages describe the functions of the various control statements.

The use examples show as comment statements the source files to which generated instructions are input.

Conditional branch — if

(1) if~elseif~else~endif

[Coding format]

```
[Δ] if [Δ] (Conditional expression 1) [Δ] [(Register name)]
      if block
[Δ] elseif [Δ] (Conditional expression 2) [Δ] [(Register name)]
      elseif block
[Δ] else
      else block
[Δ] endif
```

[Function]

<1> if~endif

The if block is executed if conditional expression 1 is true.

The if block may occupy several lines.

<2> if~else~endif

The if block is executed if conditional expression 1 is true and the else block is executed if it is false.

The if block and else block may occupy several lines.

<3> if~elseif~else~endif

Several elseif blocks can be written for a single if statement.

If conditional expression 1 is true, the if block is executed. If it is false, conditional expression 2 is tested.

If conditional expression 2 is true, the elseif block is executed. If it is false, the condition of any other elseif that exists prior to the next endif is tested. If there is no elseif, the else block is executed.

The if block, elseif block, and else block may occupy several lines.

Conditional branch	if

[Description]

<1> Comparison expressions, logic expressions, and test bit expressions can be entered in conditional expressions. If a register name is specified, the specified register is used when testing conditions.
For details of comparison expressions and logic expressions, see **"3.6 Conditional Expressions"**.

<2> if~else~endif is used when coding two branches for a condition.

<3> if~elseif~else~endif is used when coding several branches for a certain range of values. This differs from a switch statement in that the statement contains a range of values.

<4> elseif statements and else statements can be omitted and several elseif statements can be entered.

[Generated instructions]

<1> Processing of if (conditional expression)
(a) Generates an instruction to test the condition of the conditional expression.
(b) Generates a branch instruction to branch to an elseif block or else block if the condition is not met.

<2> Processing of elseif (conditional expression)
(a) Generates a branch instruction to an endif statement.
(b) Generates a label for the branch instruction generated by an if statement.
(c) Generates an instruction to test the condition of the conditional expression.
(d) Generates a branch instruction to branch to an elseif block or else block if the condition is not met.

<3> Processing of else
(a) Generates a branch instruction to an endif statement.
(b) Generates a label for the branch instruction generated by an if statement or elseif statement.

<4> Processing of endif
(a) Generates a label for the branch instruction generated by an if statement, elseif statement, or else statement.

<5> Additional description
(a) These blocks can be mixed in memory with elseif_bit.

Conditional branch **if**

[Use examples]

<1> When entered in lower case letters

```
        CMP     A,#0            ;if(A==#0)
        BNZ     $?L1
        BF      TFLG.0,$?L2     ;CY=TFLG.0
        SET1    CY
        BR      ?L3
?L2:
        CLR1    CY
?L3:
        MOVW    AX,#0FFH        ;AX=#0FFH
        BR      ?L4
?L1:                            ;else
        MOVW    BC,#0A00H       ;BC=#0A00H
?L4:                            ;endif
```

<2> When entered in upper case letters

```
        CMP     A,#0            ;IF(A==#0)
        BZ      $?L5
        BR      ?L6
?L5:
        BF      TFLG.0,$?L7     ;CY=TFLG.0
        SET1    CY
        BR      ?L8
?L7:
        CLR1    CY
?L8:
        MOVW    AX,#0FFH        ;AX=#0FFH
        BR      ?L9
?L6:                            ;ELSE
        MOVW    BC,#0A00H       ;BC=#0A00H
?L9:                            ;ENDIF
```

Conditional branch **if_bit**

(2) if_bit~elseif_bit~else~endif

[Coding format]

```
[Δ] if_bit [Δ](Conditional expression 1) [Δ][(Register name)]
      if_bit block
[Δ] elseif_bit [Δ](Conditional expression 2) [Δ][(Register name)]
      elseif_bit block
[Δ] else [Δ]
      else block
[Δ] endif [Δ]
```

[Function]

<1> if_bit~endif
If conditional expression 1 is true, the if_bit block is executed.
The if_bit block may occupy several lines.

<2> if_bit~else~endif
The if_bit block is executed if conditional expression 1 is true and the else block is executed if it is false.
The if_bit block and else block may occupy several lines.

<3> if_bit~elseif_bit~else~endif
If conditional expression 1 is true, the if_bit block is executed. If it is false, conditional expression 2 is tested.
If conditional expression 2 is true, the elseif_bit block is executed. If it is false, the condition of any elseif_bit that exists before the next endif is tested.
If there is no elseif_bit, the else block is executed.
The if_bit block, elseif_bit block, and else block may occupy several lines.

<4> Additional description
These blocks can be mixed in memory with elseif.

[Description]

<1> Test bit expressions are entered as conditional expressions 1 and 2.
For details of test bit expressions, see **"3.6 Conditional Expressions"**.

<2> if_bit~else~endif is used when coding two branches for a condition.
if_bit~elseif_bit~else~endif is used when checking several bit symbols for multiple branches.

<3> elseif_bit statements and else statements can be omitted and several elseif_bit statements can be entered.

Conditional branch	if_bit

[Generated instructions]

<1> Processing of if_bit (bit condition)

(a) Generates a true/false instruction for a bit condition.

<2> Processing of elseif_bit (bit condition)

(a) Generates a branch instruction to an endif statement.

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

(c) Generates a true/false instruction for a bit condition.

<3> Processing of else

(a) Generates a branch instruction to an endif statement.

(b) Generates a label for the branch instruction generated by an if_bit statement or elseif_bit statement.

<4> Processing of endif

(a) Generates a label for the branch instruction generated by an if_bit statement, elseif_bit statement, or else statement.

Conditional branch **if_bit**

[Use examples]

<1> When entered in lower case letters

```
        BT      TRFG.0,$?L1     ;if_bit(!TRFG.0)
        SET1    PRTYFLG.3       ;PRTYFLG.3=1
        BR      ?L2
?L1:                            ;elseif_bit(PGF.0)
        BF      PGF.0,$?L3
        MOVW    BC,#0FFH        ;BC=#0FFH
        BR      ?L2
?L3:                            ;else
        MOV     A,#(FG SHR 6)   ;H=#(FG SHR 6)(A)
        MOV     H,A
        BF      PFG.0,$?L4      ;CY=PFG.0
        SET1    CY
        BR      ?L5
?L4:
        CLR1    CY
?L5:
        CLR1    BUSYFG.2        ;BUSYFG.2=0
?L2:                            ;endif
```

<2> When entered in upper case letters

```
        BF      TRFG.0,$?L6     ;IF_BIT(!TRFG.0)
        BR      ?L7
?L6:
        SET1    PRTYFLG.3       ;PRTYFLG.3=1
        BR      ?L8
?L7:                            ;ELSEIF_BIT(PGF.O)
        BT      PGF.O,$?L9
        BR      ?L10
?L9:
        MOVW    BC,#0FFH        ;BC=#0FFH
        BR      ?L8
?L10:                           ;ELSE
        MOV     A,#(FG SHR 6)   ;H=#(FG SHR 6)(A)
        MOV     H,A
        BF      PFG.O,$?L11     ;CY=PFG.0
        SET1    CY
        BR      ?L12
?L11:
        CLR1    CY
?L12:
        CLR1    BUSYFG.2        ;BUSYFG.2=0
?L8:                            ;ENDIF
```

Conditional branch **switch**

(3) switch~case~default~ends

[Coding format]

```
[Δ] switch [Δ]([Δ] case symbol [Δ] ) [Δ][(specified register)]
      [Δ] case [Δ] Constant:
              Statement_1
[     [Δ] case [Δ] Constant:
              Statement_2
      [Δ] [default:]
              Statement_N
[Δ] ends
```

[Function]

<1> If the value of the case symbol matches the case constant, the specified statement is executed.

<2> If the value of the case symbol does not match any case constant and a default statement has been entered, the default statement is executed.

<3> Normally, a break statement must be entered to skip a switch block.

[Description]

<1> The possible specifications for “case symbol” depend on the assembly language of the target device.

<2> If a break statement is not entered, a comparison instruction is executed for the next case statement.
Note with caution that operations following case processing differ from those in C-language programs. Enter a branch instruction to establish a function similar to a C program.

<3> Constants can be expressed as binary, octal, decimal, hexadecimal, or character string constants.
However, since the structured assembler recognizes constants as character strings, be careful to use only constants that the assembler can recognize as such.

<4> The case symbol is transferred to the specified register only when a register specification has been made.

Conditional branch **switch**

[Generated instructions]

<1> Processing of switch statement

(a) If a register has not been specified, the case symbol is tested and, when necessary, a transfer instruction to A or AX is generated.

(b) If a register has been specified, the case symbol is transferred to the specified register.
However, an error occurs if a comparison instruction cannot be generated.
For details, see **Table 3-1**. **Generated Instructions for switch Statement**.

<2> Processing of case statement

(a) Labels are generated from branch processing from other case statements.

(b) CMP or CMPW is generated, and if the specified constant does not match, a branch instruction for another case statement, default statement, or ends statement is generated.
?LTRUE : Branch destination label when specified constant matches
?LFALSE : Branch destination label when specified constant does not match

- If the case statement is expressed in lower case letters and a register specification has not been made in the switch statement

```
    CMP(W)    case symbol, #case constant
    BNZ       $?LFALSE
```

- If the case statement is expressed in lower case letters and a register specification has been made in the switch statement

```
    CAMP(W)   specified register, #case constant
    BNZ       $?LFALSE
```

- If the case statement is expressed in upper case letters and a register specification has not been made in the switch statement

```
    CMAP(W)   case symbol, #case constant
    BZ        $?LTRUE
    BR        ?LFALSE
?LTRUE
```

- If the case statement is expressed in upper case letters and a register specification has been made in the switch statement

```
    CAMP(W)   specified register, #case constant
    BZ        $?LTRUE
    BR        ?LFALSE
?LTRUE
```

<3> Processing of default statement

(a) Generates a label for the branch instruction from the case statement

<4> Processing of ends statement

(a) Generates a label for the branch instruction from the case statement or break statement

Table 3-1. Generated Instructions for switch Statement

CASE symbol		Without register specification	With register specification												
			a	b	f	g	h	i	j	k	n	o	p	q	r
a	CY														
b	Bit symbol														
c	[HL]. β														
d	Byte user symbol	***3**			***1**						***2**				
e	Byte data	***3**			***1**										
f	A	***3**													
g	Byte register	***1**			***1**										
h	R0	***1**			***1**										
i	R1														
j	sfr	***1**			***1**										
k	PSW	***1**			***1**										
l	Word user symbol				***1**						***2**				
m	Word data	***2**									***2**				
n	AX	***3**													
o	Word register	***2**									***2**				
p	RP0														
q	sfrp														
r	SP	***2**									***2**				
s	Direct access symbol	***1**			***1**										
t	Indirect access symbol	***1**			***1**										
u	[DE]	***1**			***1**										
v	Immediate symbol	***1**			***1**						***2**				

***1** : Generates MOV instruction
***2** : Generates MOVW instruction
***3** : Does not generate transfer instruction
Empty columns indicate errors.

Conditional branch **switch**

[Use examples]

<1> When entered in lower case letters

```
        MOV     A,R0            ;SWITCH(R0)
        CMP     A,#1            ; case 1:
        BNZ     $?L1
        BF      P1.0,$?L2       ;  if_bit(P1.0)
        BTM.3                   ;  BTM.3
?L2:                            ;  endif
        BR      ?L3             ; break
?L1:                            ; case 2:
        CMP     A,#2
        BNZ     $?L4
        BR      ?L3             ; break
?L4:                            ; case 3:
        CMP     A,#3
        BNZ     $?L5
        BR      ?L3             ; break
?L5:                            ; default:
?L3:                            ;ENDS
```

<2> When entered in upper case letters

```
        MOV     A,R0            ;SWITCH(R0)
        CMP     A,#1            ; CASE 1:
        BZ      $?L6
        BR      ?L7
?L6:
        BF      P1.0,$?L8       ;  if_bit(P1.0)
        BTM.3                   ;  BTM.3
?L8:                            ;  endif
        BR      ?L9             ; break
?L7:                            ; CASE 2:
        CMP     A,#2
        BZ      $?L10
        BR      ?L11
?L10:
        BR      ?L9             ; break
?L11:                           ; CASE 3:
        CMP     A,#3
        BZ      $?L12
        BR      ?L13
?L12:
        BR      ?L9             ; break
?L13:                           ; DEFAULT:
?L9:                            ;ENDS
```

Conditional loop **for**

(4) for~next

[Coding format]

```
[Δ] for [Δ] ([expression 1] ; [expression 2] ; [expression 3]) [Δ][(register
specification)]
      Instruction group
[Δ] next
```

[Function]

The initial value is set by expression 1 and the statement and expression 3 are executed as long as the conditional expression in expression 2 is met. Usually, expression 3 is an increment or decrement operation.

The meaning is similar to the example shown below.

Expression 1
while (expression 2)
Instruction group
Expression 3
endw

[Description]

<1> Be sure to note that the similar example shown above does not apply to generated instructions.

<2> The following are entered in expression 1, expression 2, and expression 3.

- Expression 1 … Initial value setting (assignment expression)
- Expression 2 … Conditional expression
- Expression 3 … Increment or decrement expression

<3> Assignment operators and exchange statements can be entered in expression 1 or expression 3, but when doing so, the conversion output should be checked and modified if necessary.

<4> It is possible to omit expression 1, expression 2, or expression 3. However, if expression 2 is omitted, an endless loop will occur.

<5> “forever” can be entered in a conditional expression.

<6> Since expression 2 and expression 3 control for~next, the contents of these expressions should not be changed by an executable statement. Changing these contents can result in faulty operation.

Conditional loop	for

[Generated instructions]

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

(a) Generates instruction for expression 1. If a register name has been specified, the specified register is used for assignments and comparisons.

(b) Generates a branch instruction to the statement that tests expression 2's conditions.

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

(d) Generates a label for the branch instruction generated at (2).

(e) Generates a condition testing instruction for expression 2.

<2> Processing of next statement

(a) Generates a branch instruction to the label generated via for statement processing (3).

(b) Generates a label for the branch instruction for skipping a for block.

(c) Generates an instruction for expression 3's assignment expression.

<3> Additional description

(a) The following method is recommended for more effective use of for~next statements.

1. Use saddr instead of a register name as the control variable in expression 1 and expression 3.
2. When specifying a register, specify either A or AX.
3. When executing a loop for at least 256 repetitions, nest a for statement and use two saddr variables as the control variables.

Remark The above method is recommended because of the limited range of symbols that can be entered as operands in order to output CMP or CMPW as generated instructions for the conditional expression in expression 2.

Conditional loop **for**

[Use examples]

<1> When entered in lower case letters

```
          MOV     i,#0H    ;for(i=#0H;i<#0FFH;i++)
?L1:
          CMP     i,#0FFH
          BNC     $?L2
          CALL    !XXX     ; CALL  !XXX
          INC     i
          BR      ?L1
?L2:                       ;next
```

<2> When entered in upper case letters

```
          MOV     i,#0H    ;FOR(i=#0H;i<#0FFH;i++)
?L3:
          CMP     i,#0FFH
          BC      $?L4
          BR      ?L5
?L4:
          CALL    !XXX     ; CALL  !XXX
          INC     i
          BR      ?L3
?L5:                       ;NEXT
```

Conditional loop **while**

(5) while~endw

[Coding format]

```
[Δ] while [Δ] (conditional expression) [Δ] [(register specification)]
      Instruction group
[Δ] endw
```

[Function]

<1> The instruction group is repeatedly executed as long as the conditional expression remains true.

[Description]

<1> It is possible to enter comparison expressions, logic expressions, test bit expressions, and “forever” as conditional expressions.
If “forever” is entered, the result is an endless loop.

<2> As the register name, specify the register used in the comparison expression or logic expression entered as “(conditional expression)”.

<3> Since the conditional expression is tested before the instruction group is executed, if the first conditional expression is found to be false, the instruction group is not executed even once.

[Generated instructions]

<1> Processing of while (conditional expression) statement

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

(b) Generates a condition testing instruction. If a register name has been specified, the specified register is used when generating the condition testing instruction.

(c) Generates a branch instruction for removing the while (conditional expression) statement from the while block when the condition tests as false.

<2> endw

(a) Generates a branch instruction for an execution loop.

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

Conditional loop **while**

[Use examples]

<1> When entered in lower case letters

```
?L1:                                ;while(AX<#0FFFH)
        CMPW    AX,#0FFFH
        BNC     $?L2
        MOV     B,#0FH              ; B=#0FH
        INCW    HL                  ; HL++
        BR      ?L1
?L2:                                ;endw
```

<2> When entered in upper case letters

```
?L3:                                ;WHILE(AX<#0FFFH)
        CMPW    AX,#0FFFH
        BC      $?L4
        BR      ?L5
?L4:
        MOV     B,#0FH              ; B=#0FH
        INCW    HL                  ; HL++
        BR      ?L3
?L5:                                ;ENDW
```

Conditional loop **while_bit**

(6) while_bit~endw

[Coding format]

```
[Δ] while_bit [Δ] (bit condition)
      Instruction group
[Δ] endw
```

[Function]

<1> The instruction group can be executed as long as the bit condition is true.

[Description]

<1> Since the bit condition is tested before the instruction group is executed, if the first bit condition is found to be false, the instruction group is not executed even once.

[Generated instructions]

<1> Processing of while_bit (bit condition) statement

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

(b) Generates an instruction for testing the bit condition as true or false.

(c) Generates a branch instruction for removing the while_bit statement from the while_bit~endw block when the bit condition tests as false.

<2> Processing of endw

(a) Generates a branch instruction for an execution loop.

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

Conditional loop **while_bit**

[Use examples]

<1> When entered in lower case letters

```
?L1:                                ;while_bit(!TRFG.0)
         BT      TRFG.0,$?L2
         MOV     A,PORT1            ; A=PORT1
         CMP     A,#04H             ; if(A==#04H)
         BNZ     $?L3
         MOV     X,#0FFH            ;  X=#0FFH
         BR      ?L4
?L3:                                ; else
         CLR1    PFG.0              ;  PFG.0=0
?L4:                                ; endif
         BR      ?L1
?L2:                                ;endw
```

<2> When entered in upper case letters

```
?L5:                                ;WHILE_BIT(!TRFG.0)
         BF      TRFG.0,$?L6
         BR      ?L7
?L6:
         MOV     A,PORT1            ; A=PORT1
         CMP     A,#04H             ; if(A==#04H)
         BNZ     $?L8
         MOV     X,#0FFH            ;  X=#0FFH
         BR      ?L9
?L8:                                ; else
?L9:                                ; endif
         BR      ?L5
?L7:                                ;ENDW
```

Conditional loop **until**

(7) repeat~until

[Coding format]

```
[Δ] repeat
      Instruction group
[Δ] until [Δ] (conditional expression) [Δ] [(register specification)]
```

[Function]

<1> The instruction group is repeatedly executed as long as the conditional expression remains true.

[Description]

<1> It is possible to enter comparison expressions, logic expressions, test bit expressions, and “forever” as conditional expressions.
If “forever” is entered, the result is an endless loop.

<2> As the register name, specify the register used in the comparison expression or logic expression entered as “(conditional expression)”.

<3> The conditional expression is tested after the instruction group is executed. Therefore, if the first conditional expression is found to be true, the instruction group is executed once.

[Generated instructions]

<1> Processing of repeat statement

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

<2> Processing of until (conditional expression) statement

(a) Generates a condition testing instruction for the conditional expression.

(b) Generates a branch instruction for the label that was generated by repeat in order to execution the instruction group during repeat~until and while the conditional expression tests as false. If the conditional expression tests as true, the until statement is removed from the repeat block.

Conditional loop **until**

[Use examples]

<1> When entered in lower case letters

```
?L1:                                ;repeat
         MOVW    AX,BC              ; AX=BC
         CMP     ABC,#0CH           ; if(ABC==#0CH)
         BNZ     $?L2
         CALL    !XXX               ;  CALL !XXX
?L2:                                ; endif
         INC     CNT                ; CNT++
         CMP     CNT,#0FFH          ;until(CNT==#0FFH)
         BNZ     $?L1
```

<2> When entered in upper case letters

```
?L3:                                ;REPEAT
         MOVW    AX,BC              ; AX=BC
         CMP     ABC,#0CH           ; if(ABC==#0CH)
         BNZ     $?L4
         CALL    !XXX               ;  CALL !XXX
?L4:                                ; endif
         INC     CNT                ; CNT++
         CMP     CNT,#0FFH          ;UNTIL(CNT==#0FFH)
         BZ      $?L5
         BR      ?L3
?L5:
```

Conditional loop **until_bit**

(8) until_bit

[Coding format]

```
[Δ] repeat
      Instruction group
[Δ] until_bit [Δ] (test bit expression)
```

[Function]

<1> The instruction group is repeatedly executed as long as the bit condition is false.

[Description]

<1> The bit condition is tested after the instruction group is executed. Therefore, if the first bit condition is found to be true, the instruction group is executed once.

[Generated instructions]

<1> Processing of repeat

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

<2> Processing of until_bit (bit condition)

(a) Generates a branch instruction for the label that is generated by repeat in order to execute the instruction group between repeat and until_bit when the conditional expression tests as false. If the conditional expression tests as true, until_bit is removed from the repeat block.

[Use examples]

<1> When entered in lower case letters

```
?L1:                                ;repeat
          MOV     B,#8H             ; B=#8H
          CALL    !XXX              ; CALL  !XXX
          BF      TRFG.0,$?L1       ;until_bit(TRFG.0)
```

<2> When entered in upper case letters

```
?L2:                                ;REPEAT
          MOV     B,#8H             ; B=#8H
          CALL    !XXX              ; CALL  !XXX
          BT      TRFG.0,$?L3       ;UNTIL_BIT(TRFG.0)
          BR      ?L2
?L3:
```

(9) break

[Coding format]

```
[Δ] break
```

[Function]

Terminates execution of the innermost nested block among while, repeat, for, and switch blocks.

[Description]

An error occurs if a statement other than a while, while_bit, repeat~until, repeat~until_bit, for, or switch statement has been entered.

[Generated instructions]

Generates an unconditional branch instruction to remove while, repeat, for, or switch blocks.

BR ?Lxxxx

[Use example]

```
?L1:                                ;while(forever)
        MOV     X,#0                ; X=#0
        MOV     PORT4,A             ; PORT4=A
        CMP     A,#0FH              ; if(A==#0FH)
        BNZ     $?L2
        BR      ?L3                 ;  break
?L2:                                ; endif
        INCW    HL                  ; HL++
        BR      ?L1
?L3:                                ;endw
```

Conditional loop **continue**

(10) Continue

[Coding format]

```
[Δ] continue
```

[Function]

Skips processing following continue within the innermost nested block among a while, while_bit, repeat~until, repeat~until_bit, or for statement and sets an unconditional branch before the condition is tested.

[Description]

<1> This is used to skip subsequent processing from the middle of a block and execute the next loop.

<2> An error occurs if a statement other than a while, while_bit, repeat~until, repeat~until_bit, or for statement has been entered.

[Generated instructions]

Generates an unconditional branch instruction for a label to repeat a while, while_bit, repeat~until, repeat~until_bit, or for block.

BR ?Lxxxx

[Use example]

```
?L1:                                ;while(SYM==#0FH)
        CMP     SYM,#0FH
        BNZ     $?L2
        MOV     B,#0                ; B=#0
        MOV     PORT4,A             ; PORT4=A
        CMP     A,#0FH              ; if(A==#0FH)
        BNZ     $?L3
        BR      ?L1                 ;  continue
        BR      ?L4
?L3:                                ; else
        INCW    HL                  ;  HL++
?L4:                                ; endif
        BR      ?L1
?L2:                                ;endw
```

(11) goto

[Coding format]

```
[Δ] goto Δ label
```

[Function]

Unconditionally branches to a label.

[Description]

<1> goto statements are entered when immediate error processing is required such as in an error processing program, or when collective processing of errors at multiple locations is needed.

<2> The symbols shown in the assembly language label column are specified as label names.

[Generated instructions]

<1> Generates the following instruction.

BR Label

<2> The goto statement's labels are not automatically generated by the structured assembler. Note also that the assembler does not automatically check whether or not a branch destination label exists.

[Use examples]

```
?L1:                                ;while(forever)
          MOV     B,#0              ; B=#0
          MOV     PORT4,A           ; PORT4=A
          CMP     A,#0FH            ; if(A==#0FH)
          BNZ     $?L2
          BR      ERROR             ;  goto ERROR
?L2:                                ; endif
          INCW    HL                ; HL++
          BR      ?L1
                                    ;endw
```

3.6 Conditional Expressions

Conditional expressions are used to set conditions via control statements.
The following are examples of conditional expressions.

- Comparison expression ··· Compares first and second values and tests them as true or false.
- Test bit expression ····· Determines flag on/off status based on bit symbols.
- Logical operation ····· Performs a logical operation for a conditional expression when conditions are combined.

Table 3-2. Comparison Expressions

Comparison expression		Coding format	Function
(1)	Equal	α == β	True when $\alpha = \beta$, false when $\alpha \neq \beta$
(2)	NotEqual	α != β	True when $\alpha \neq \beta$, false when $\alpha = \beta$
(3)	LessThan	$\alpha < \beta$	True when $\alpha < \beta$, false when $\alpha >= \beta$
(4)	GreaterThan	$\alpha > \beta$	True when $\alpha > \beta$, false when $\alpha <= \beta$
(5)	GreaterEqual	$\alpha >= \beta$	True when $\alpha >= \beta$, false when $\alpha < \beta$
(6)	LessEqual	$\alpha <= \beta$	True when $\alpha <= \beta$, false when $\alpha > \beta$

Table 3-3. Test Bit Expressions

Test bit expression		Coding format	Function
(7)	Positive logic (bit)	Bit symbol	True when specified bit value is 1
(8)	Negative logic (bit)	!bit symbol	True when specified bit value is 0

Table 3-4. Logical Operations

Logical operation		Coding format	Function
(9)	Logical AND	Conditional expression 1 && conditional expression 2	True if both conditional expression 1 and conditional expression 2 are true
(10)	Logical OR	Conditional expression 1 || conditional expression 2	True if either conditional expression 1 or conditional expression 2 is true

If (γ) is specified at the end of a comparison, a comparison can be made between α and β values that cannot be compared directly.

γ specifies the register that is used for this comparison.

3.6.1 Comparison expressions

In the description of each comparison expression, “?LTRUE” is used as the branch destination label for when the comparison tests as true and ?LFALSE is used as the branch destination label when it tests as false.

See “**3.4 Register Specification**” for a description of the register specification coding format.

The structured assembler does not test whether or not the symbols entered on the left and right sides of a comparison expression are entered correctly as assembly language operands. However, a data size test is performed, as described in “**2.6 Data Sizes**” to determine whether or not an instruction can be generated. In addition, when specifying a register, the possibility of generating an instruction using the specified register is tested.

An error message is output when a test results in an error.

For details, see the relevant generated instruction.

The various comparison expressions are described below.

Table 3-5. Generated instructions for Comparison Instructions

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY																						
	b	Bit symbol																						
	c	[HL]. β																						
	d	Byte user symbol																						***1**
	e	Byte data																						***1**
	f	A				***1**	***1**		***1**	***1**	***1**			***1**							***1**	***1**		***1**
	g	Byte register																						
	h	R0																						
	i	R1																						
αn	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX																						***2**
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1** : Generates CMP instruction

***2** : Generates CMPW instruction

Empty columns indicate errors.

Comparison expressions **Equal (==)**

(1) Equal (==)

[Coding format]

```
[Δ] [size specification] α [Δ] == [Δ] [size specification] [Δ] β [Δ] [(register
specification)]
```

[Function]

<1> When there is no register specification

True when the contents of α and β are equal, false when they are not equal.

<2> When there is a register specification

The contents of α are transferred to the specified register. True is the result when the contents of the specified register are equal to the contents of β and false is the result when they are not equal.

[Description]

<1> When there is no register specification

For α and β, be sure to specify contents that can be entered in CMP or CMPW.

<2> When there is a register specification

For α, be sure to specify contents that can be entered in MOV or MOVW.

For β, be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

<1> If the control statement is entered in lower case letters and there is no register specification

CMP(W)	α , β
BNZ	$?LFALSE

<2> If the control statement is entered in lower case letters and there is a register specification

MOV(W)	Specified register, α
CMP(W)	Specified register, β
BNZ	$?LFALSE

Comparison expressions **Equal (==)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
    CMP(W)    α, β
    BZ        $?LTRUE
    BR        ?LFALSE
?LTRUE:
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
    MOV(W)    Specified register, α
    CMP(W)    Specified register, β
    BZ        $?LTRUE
    BR        ?LFALSE
?LTRUE:
```

For details of combinations of α and β, see **Table 3-5. Generated instructions for Comparison Instructions**. α indicates the specified register. For further description of generated instructions for MOV, see **"CHAPTER 4 (1) Assign"**.

[Use examples]

<1> If the control statement is entered in lower case letters and there is no register specification

```
        CMPW    AX,#0F0FH       ;if(AX==#0F0FH)
        BNZ     $?L1
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif
```

<2> If the control statement is entered in lower case letters and there is a register specification

```
        MOV     A,!XYZ          ;if(!XYZ==#5(A))
        CMP     A,#5
        BNZ     $?L3
        CALL    !PPP            ; CALL  !PPP
?L3:                            ;endif
```

Comparison expressions **Equal (==)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
        CMPW    AX,#0F0FH       ;IF(AX==#0F0FH)
        BZ      $?L4
        BR      ?L5
?L4:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;ENDIF
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
        MOV     A,!XYZ          ;IF(!XYZ==#5(A))
        CMP     A,#5
        BZ      $?L7
        BR      ?L8
?L7:
        CALL    !PPP            ; CALL  !PPP
?L8:                            ;ENDIF
```

(2) NotEqual (!=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] != [Δ] [size specification] [Δ] β [Δ] [(register
specification)]
```

[Function]

<1> When there is no register specification

True when the contents of α and β are not equal, false when they are equal.

<2> When there is a register specification

The contents of α are transferred to the specified register. True is the result when the contents of the specified register are not equal to the contents of β and false is the result when they are equal.

[Description]

<1> When there is no register specification

For α and β, be sure to specify contents that can be entered in CMP or CMPW.

<2> When there is a register specification

For α, specify contents that can be entered in MOV or MOVW.

For β, specify contents that can be entered in CMP or CMPW.

[Generated instructions]

<1> If the control statement is entered in lower case letters and there is no register specification

CMP(W)	α, β
BZ	$?LFALSE

<2> If the control statement is entered in lower case letters and there is a register specification

MOV(W)	Specified register, α
CMP(W)	Specified register, β
BZ	$?LFALSE

Comparison expressions **NotEqual (!=)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
    CMP(W)    α, β
    BNZ       $?LTRUE
    BR        ?LFALSE
?LTRUE:
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
    MOV(W)    Specified register, α
    CMP(W)    Specified register, β
    BNZ       $?LTRUE
    BR        ?LFALSE
?LTRUE:
```

For details of combinations of α and β, see **Table 3-5. Generated instructions for Comparison Instructions**. α indicates the specified register. For further description of generated instructions for MOV, see **"CHAPTER 4 (1) Assign"**.

[Use examples]

<1> If the control statement is entered in lower case letters and there is no register specification

```
          CMPW    AX,#0FFFH       ;if(AX!=#0FFFH)
          BZ      $?L1
          CALL    !XXX            ; CALL  !XXX
          BR      ?L2
?L1:                              ;else
          CALL    !YYY            ; CALL  !YYY
?L2:                              ;endif
```

<2> If the control statement is entered in lower case letters and there is a register specification

```
          MOV     A,!XYZ          ;if(!XYZ!=#5(A))
          CMP     A,#5
          BZ      $?L3
          CALL    !PPP            ; CALL  !PPP
?L3:                              ;endif
```

Comparison expressions **NotEqual (!=)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
        CMPW    AX,#0FFFH       ;IF(AX!=#0FFFH)
        BNZ     $?L4
        BR      ?L5
?L4:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;ENDIF
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
        MOV     A,!XYZ          ;IF(!XYZ!=#5(A))
        CMP     A,#5
        BNZ     $?L7
        BR      ?L8
?L7:
        CALL    !PPP            ; CALL  !PPP
?L8:                            ;ENDIF
```

(3) LessThan (<)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] < [Δ] [size specification] [Δ] β [Δ] [(register
specification)]
```

[Function]

<1> When there is no register specification

True when the contents of α are less than the contents of β, false when otherwise (i.e., equal to or greater than).

<2> When there is a register specification

The contents of α are transferred to the specified register. True is the result when the contents of the specified register are less than the contents of β and false is the result when they are otherwise.

[Description]

<1> When there is no register specification

For α and β, be sure to specify contents that can be entered in CMP or CMPW.

<2> When there is a register specification

For α, be sure to specify contents that can be entered in MOV or MOVW.

For β, be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

<1> If the control statement is entered in lower case letters and there is no register specification

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

<2> If the control statement is entered in lower case letters and there is a register specification

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

<3> If the control statement is entered in upper case letters and there is no register specification

```
    CMP(W)      α, β
    BC          $?LTRUE
    BR          ?LFALSE
?LTRUE:
```

Comparison expressions **LessThan (<)**

<4> If the control statement is entered in upper case letters and there is a register specification

```
    MOV(W)      Specified register, α
    CMP(W)      Specified register, β
    BC          $?LTRUE
    BR          ?LFALSE
?LTRUE:
```

For details of combinations of α and β, see **Table 3-5. Generated instructions for Comparison Instructions**. α indicates the specified register. For further description of generated instructions for MOV, see **"CHAPTER 4 (1) Assign"**.

[Use examples]

<1> If the control statement is entered in lower case letters and there is no register specification

```
        CMP     A,[HL]          ;if(A<[HL])
        BNC     $?L1
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif
```

<2> If the control statement is entered in lower case letters and there is a register specification

```
        MOVW    AX,ABCP         ;if(ABCP<#0FE00H(AX))
        CMPW    AX,#0FE00H
        BNC     $?L3
        CALL    !PPP            ; CALL  !PPP
?L3:                            ;endif
```

<3> If the control statement is entered in upper case letters and there is no register specification

```
        CMP     A,[HL]          ;IF(A<[HL])
        BC      $?L4
        BR      ?L5
?L4:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;ENDIF
```

Comparison expressions **LessThan (<)**

<4> If the control statement is entered in upper case letters and there is a register specification

```
        MOVW    AX,ABCP         ;IF(ABCP<#0FE00H(AX))
        CMPW    AX,#0FE00H
        BC      $?L7
        BR      ?L8
?L7:
        CALL    !PPP            ; CALL  !PPP
?L8:                            ;ENDIF
```

(4) GreaterThan (>)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] > [Δ] [size specification] [Δ] β [Δ] [(register
specification)]
```

[Function]

<1> When there is no register specification

True when the contents of α are greater than the contents of β, false when otherwise (i.e. equal to or less than).

<2> When there is a register specification

The contents of α are transferred to the specified register. True is the result when the contents of the specified register are greater than the contents of β and false is the result when they are otherwise.

[Description]

<1> When there is no register specification

For α and β, be sure to specify contents that can be entered in CMP or CMPW.

<2> When there is a register specification

For α, be sure to specify contents that can be entered in MOV or MOVW.

For β, be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

<1> If the control statement is entered in lower case letters and there is no register specification

CMP(W)	α, β
BZ	$?LFALSE
BC	$?LFALSE

<2> If the control statement is entered in lower case letters and there is a register specification

MOV(W)	Specified register, α
CMP(W)	Specified register, β
BZ	$?LFALSE
BC	$?LFALSE

Comparison expressions **GreaterThan (>)**

<3> If the control statement is entered in upper case letters and there is no register specification

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

<4> If the control statement is entered in upper case letters and there is a register specification

```
    MOV(W)      Specified register, α
    CMP(W)      Specified register, β
    BZ      $$+4
    BNC         $?LTRUE
    BR          ?LFALSE
?LTRUE:
```

For details of combinations of α and β, see **Table 3-5. Generated instructions for Comparison Instructions**. α indicates the specified register. For further description of generated instructions for MOV, see **"CHAPTER 4 (1) Assign"**.

[Use examples]

<1> If the control statement is entered in lower case letters and there is no register specification

```
        CMP     A,[HL]          ;if(A>[HL])
        BZ      $?L1
        BC      $?L1
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif
```

<2> If the control statement is entered in lower case letters and there is a register specification

```
        MOVW    AX,ABCP         ;if(ABCP>#0FE40H(AX))
        CMPW    AX,#0FE40H
        BZ      $?L3
        BC      $?L3
        CALL    !PPP            ; CALL  !PPP
?L3:                            ;endif
```

Comparison expressions **GreaterThan (>)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
        CMP     A,[HL]          ;IF(A>[HL])
        BZ      $$+4
        BNC     $?L4
        BR      ?L5
?L4:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;ENDIF
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
        MOVW    AX,ABCP         ;IF(ABCP>#0FE40H(AX))
        CMPW    AX,#0FE40H
        BZ      $$+4
        BNC     $?L7
        BR      ?L8
?L7:
        CALL    !PPP            ; CALL  !PPP
?L8:                            ;ENDIF
```

(5) GreaterEqual (>=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] >= [Δ] [size specification] [Δ] β [Δ] [(register
specification)]
```

[Function]

<1> When there is no register specification

True when the contents of α are greater than or equal to the contents of β, false when they are less than the contents of β.

<2> When there is a register specification

The contents of α are transferred to the specified register. True is the result when the contents of the specified register are greater than or equal to the contents of β and false is the result when they are less than the contents of β.

[Description]

<1> When there is no register specification

For α and β, be sure to specify contents that can be entered in CMP or CMPW.

<2> When there is a register specification

For α, be sure to specify contents that can be entered in MOV or MOVW.

For β, be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

<1> If the control statement is entered in lower case letters and there is no register specification

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

<2> If the control statement is entered in lower case letters and there is a register specification

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

<3> If the control statement is entered in upper case letters and there is no register specification

```
    CMP(W)      α, β
    BNC         $?LTRUE
    BR          ?LFALSE
?LTRUE:
```

Comparison expressions **GreaterEqual (>=)**

<4> If the control statement is entered in upper case letters and there is a register specification

```
    MOV(W)      Specified register, α
    CMP(W)      Specified register, β
    BNC         $?LTRUE
    BR          ?LFALSE
?LTRUE:
```

For details of combinations of α and β, see **Table 3-5. Generated Instructions for Comparison Instructions**. α indicates the specified register. For further description of generated instructions for MOV, see **"CHAPTER 4 (1) Assign"**.

[Use examples]

<1> If the control statement is entered in lower case letters and there is no register specification

```
        CMP     A,[HL]          ;if(A>=[HL])
        BC      $?L1
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif
```

<2> If the control statement is entered in lower case letters and there is a register specification

```
        MOVW    AX,DE           ;if(DE>=#0FE30H(AX))
        CMPW    AX,#0FE30H
        BC      $?L3
        CALL    !PPP            ; CALL  !PPP
?L3:                            ;endif
```

<3> If the control statement is entered in upper case letters and there is no register specification

```
        CMP     A,[HL]          ;IF(A>=[HL])
        BNC     $?L4
        BR      ?L5
?L4:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;ENDIF
```

Comparison expressions **GreaterEqual (>=)**

<4> If the control statement is entered in upper case letters and there is a register specification

```
            MOVW    AX,DE           ;IF(DE>=#0FE30H(AX))
            CMPW    AX,#0FE30H
            BNC     $?L7
            BR      ?L8
    ?L7:
            CALL    !PPP            ; CALL  !PPP
    ?L8:                            ;ENDIF
```

(6) LessEqual (<=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] <= [Δ] [size specification] [Δ] β [Δ] [(register
specification)]
```

[Function]

<1> When there is no register specification

True when the contents of α are less than or equal to the contents of β, false when they are greater than the contents of β.

<2> When there is a register specification

The contents of α are transferred to the specified register. True is the result when the contents of the specified register are less than or equal to the contents of β and false is the result when they are greater than the contents of β.

[Description]

<1> When there is no register specification

For α and β, be sure to specify contents that can be entered in CMP or CMPW.

<2> When there is a register specification

For α, be sure to specify contents that can be entered in MOV or MOVW.

For β, be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

<1> If the control statement is entered in lower case letters and there is no register specification

CMP(W)	α, β
BZ	$$+4
BNC	$?LFALSE

<2> If the control statement is entered in lower case letters and there is a register specification

MOV(W)	Specified register, α
CMP(W)	Specified register, β
BZ	$$+4
BNC	$?LFALSE

Comparison expressions **LessEqual (<=)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
    CMP(W)    α, β
    BZ        $?LTRUE
    BC        $?LTRUE
    BR        ?LFALSE
?LTRUE:
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
    MOV(W)    Specified register, α
    CMP(W)    Specified register, β
    BZ        $?LTRUE
    BC        $?LTRUE
    BR        ?LFALSE
?LTRUE:
```

For details of combinations of α and β, see **Table 3-5. Generated Instructions for Comparison Instructions**. α indicates the specified register. For further description of generated instructions for MOV, see **"CHAPTER 4 (1) Assign"**.

[Use examples]

<1> If the control statement is entered in lower case letters and there is no register specification

```
          CMP     A,[HL]          ;if(A<=[HL])
          BZ      $$+4
          BNC     $?L1
          CALL    !XXX            ; CALL  !XXX
          BR      ?L2
?L1:                              ;else
          CALL    !YYY            ; CALL  !YYY
?L2:                              ;endif
```

<2> If the control statement is entered in lower case letters and there is a register specification

```
          MOVW    AX,HL           ;if(HL<=#0FE20H(AX))
          CMPW    AX,#0FE20H
          BZ      $$+4
          BNC     $?L3
          CALL    !PPP            ; CALL  !PPP
?L3:                              ;endif
```

Comparison expressions **LessEqual (<=)**

<3> If the control statement is entered in upper case letters and there is no register specification

```
        CMP     A,[HL]          ;IF(A<=[HL])
        BZ      $?L4
        BC      $?L4
        BR      ?L5
?L4:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;ENDIF
```

<4> If the control statement is entered in upper case letters and there is a register specification

```
        MOVW    AX,HL           ;IF(HL<=#0FE20H(AX))
        CMPW    AX,#0FE20H
        BZ      $?L7
        BC      $?L7
        BR      ?L8
?L7:
        CALL    !PPP            ; CALL  !PPP
?L8:                            ;ENDIF
```

(7) FOREVER (forever)

[Coding format]

```
[Δ] forever [Δ]
```

[Function]

Sets loop statement as an endless loop, without generating a compare instruction.

[Description]

Can be entered in a loop statement (for statement, while statement, until statement) type of conditional expression.

[Use examples]

<1> for statement

```
            MOV     i,#0            ;for(i=#0;forever;i++)
?L1:
            MOV     A,i             ; A=i
            CALL    !XXX            ; CALL  !XXX
            CMPW    AX,#0FFH        ; if(AX==#0FFH)
            BNZ     $?L2
            BR      ?L3             ;  break
?L2:                                ; endif
            INC     i
            BR      ?L1
?L3:                                ;next
```

<2> while statement

```
?L4:                                ;while(forever)
            BF      forever,$?L5
            MOV     A,i             ; A=i
            CAll    !XXX            ; CAll  !XXX
            CMPW    AX,#0FFH        ; if(AX==#0FFH)
            BNZ     $?L6
            BR      ?L5             ;  break
?L6:                                ; endif
            INC     i               ; i++
            BR      ?L4
?L5:                                ;endw
```

<3> repeat statement

```
?L7:                                ;repeat
          MOV     A,i               ; A=i
          CALL    !XXX              ; CALL  !XXX
          CMPW    AX,#0FFH          ; if(AX==#0FFH)
          BNZ     $?L8
          BR      ?L9               ;  break
?L8:                                ; endif
          INC     i                 ; i++
          BR      ?L7
?L9:                                ;until(forever)
```

3.6.2 Test bit expressions

In the description of each type of test bit expression, it is noted that ?LTRUE is used as the branch destination label when the test result is true and ?LFALSE is used as this label when the test result is false.

The structured assembler does not test whether or not test bit expression code is entered correctly as assembly language operands. However, a data size test is performed, as described in "**2.6 Data Sizes**".

In addition, "Z" is also processed as a bit symbol.

The structured assembler does not use the assembler's directive (EQU) to check whether or not a bit symbol has been defined. However, user symbols can also be processed as bit symbols.

An error message is output when the test result is an error.

For details, see the particular generating instruction.

The various test bit expressions are described below.

Test bit expressions **Positive logic (bit)**

(1) Bit symbol

[Coding format]

```
[Δ] bit symbol [Δ]
```

[Function]

True when the bit symbol contents are 1, false when they are 0.

The following control statements are able to include bit symbols entered as conditional expressions.

if	if_bit
elseif	elseif_bit
while	while_bit
until	until_bit

[Generated instructions]

<1> When the control statement is entered in lower case letters and CY has been entered

```
BNC     $?LFALSE
```

<2> When the control statement is entered in lower case letters and Z has been entered

```
BNZ     $?LFALSE
```

<3> When the control statement is entered in lower case letters and a bit symbol has been entered

```
BF      Bit symbol, $?LFALSE
```

<4> When the control statement is entered in upper case letters and CY has been entered

```
    BC      $?LTRUE
    BR      ?LFALSE
?LTRUE:
```

<5> When the control statement is entered in upper case letters and Z has been entered

```
    BZ      $?LTRUE
    BR      ?LFALSE
?LTRUE:
```

Test bit expressions **Positive logic (bit)**

<6> When the control statement is entered in upper case letters and a bit symbol has been entered.

```
    BT      Bit symbol, $?LTRUE
    BR      ?LFALSE
?LTRUE:
```

[Use examples]

<1> When the control statement is entered in lower case letters

```
        BNC     $?L1            ;if_bit(CY)
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif

        BNZ     $?L3            ;if_bit(Z)
        CALL    !XXX            ; CALL  !XXX
        BR      ?L4
?L3:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L4:                            ;endif

        BF      TRFG.0,$?L5     ;if_bit(TRFG.0)
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;endif
```

Test bit expressions **Positive logic (bit)**

<2> When the control statement is entered in upper case letters

```
        BC      $?L7            ;IF_BIT(CY)
        BR      ?L8
?L7:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L9
?L8:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L9:                            ;ENDIF

        BZ      $?L10           ;IF_BIT(Z)
        BR      ?L11
?L10:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L12
?L11:                           ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L12:                           ;ENDIF

        BT      TRFG.0,$?L13    ;IF_BIT(TRFG.0)
        BR      ?L14
?L13:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L15
?L14:                           ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L15:                           ;ENDIF
```

Test bit expressions **Negative logic (bit)**

(2) !bit symbol

[Coding format]

```
[Δ] !bit symbol [Δ]
```

[Function]

True when the bit symbol contents are 0, false when they are 1.

The following control statements are able to include bit symbols entered as conditional expressions.

if	if_bit
elseif	elseif_bit
while	while_bit
until	until_bit

[Generated instructions]

<1> When the control statement is entered in lower case letters and CY has been entered

```
BC          $?LFALSE
```

<2> When the control statement is entered in lower case letters and Z has been entered

```
BZ          $?LFALSE
```

<3> When the control statement is entered in lower case letters and a bit symbol has been entered

```
BT          Bit symbol, $?LFALSE
```

<4> When the control statement is entered in upper case letters and CY has been entered

```
    BNC     $?LTRUE
    BR      ?LFALSE
?LTRUE:
```

<5> When the control statement is entered in upper case letters and Z has been entered

```
    BNZ     $?LTRUE
    BR      ?LFALSE
?LTRUE:
```

Test bit expressions **Negative logic (bit)**

<6> When the control statement is entered in upper case letters and a bit symbol has been entered.

```
    BF      Bit symbol, $?LTRUE
    BR      ?LFALSE
?LTRUE:
```

[Use examples]

<1> When the control statement is entered in lower case letters

```
        BC      $?L1            ;if_bit(!CY)
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif

        BZ      $?L3            ;if_bit(!Z)
        CALL    !XXX            ; CALL  !XXX
        BR      ?L4
?L3:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L4:                            ;endif

        BT      TRFG.0,$?L5     ;if_bit(!TRFG.0)
        CALL    !XXX            ; CALL  !XXX
        BR      ?L6
?L5:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L6:                            ;endif
```

Test bit expressions **Negative logic (bit)**

<2> When the control statement is entered in upper case letters

```
        BNC     $?L7            ;IF_BIT(!CY)
        BR      ?L8
?L7:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L9
?L8:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L9:                            ;ENDIF

        BNZ     $?L10           ;IF_BIT(!Z)
        BR      ?L11
?L10:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L12
?L11:                           ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L12:                           ;ENDIF

        BF      TRFG.0,$?L13    ;IF_BIT(!TRFG.0)
        BR      ?L14
?L13:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L15
?L14:                           ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L15:                           ;ENDIF
```

3.6.3 Logical operations

In the description of each type of conditional expression, it is noted that ?LTRUE is used as the branch destination label when the test result is true and ?LFALSE is used as this label when the test result is false.

A logical AND (&&) or logical OR (||) result can be obtained when there are two comparison expressions or a true/false test bit expression.

Up to 16 logical operators can be entered in a conditional expression.

This means that it is possible to enter expressions for processing that is executed when two conditional expressions are both met or when either of them are met.

The structured assembler generates branch instructions beginning from the highest-priority logical operator.

[Code example]

```
B<#0FFH && C>=#0 || D==#10
```

The logical operations are described below.

(1) Logical AND (&&)

[Coding format]

```
Conditional expression 1 [Δ] && [Δ] Conditional expression 2
```

[Function]

The logical AND result of conditional expression 1 and conditional expression 2 is obtained. The result is true when conditional expression 1 and conditional expression 2 are both true and the result is false otherwise. The entered operation is performed when two conditions are met.

The output instruction differs depending on whether the control statement is entered in lower case letters or upper case letters.

Instructions for testing are generated first for contents enclosed in parentheses “( )”.

[Generated instructions]

<1> When the control statement is entered in lower case letters

Table 3-6. Generated Instructions (Control Statement in Lower Case Letters) for Logical AND

Conditional expression	Generated instruction	
α == β &&	CMP(W) BNZ	α, β $?LFALSE
α != β &&	CMP(W) BZ	α, β $?LFALSE
α < β &&	CMP(W) BNC	α, β $?LFALSE
α > β &&	CMP(W) BZ BC	α, β $?LFALSE $?LFALSE
α >= β &&	CMP(W) BC	α, β $?LFALSE
α <= β &&	CMP(W) BZ BNZ	α, β $$+4 $?LFALSE
Bit symbol &&	BF	Bit symbol, $?LFALSE
CY &&	BNC	$?LFALSE
Z &&	BNZ	$?LFALSE
!bit symbol &&	BT	Bit symbol, $?LFALSE
!CY &&	BC	$?LFALSE
!Z &&	BZ	$?LFALSE

<2> When the control statement is entered in upper case letters

Table 3-7. Generated Instructions (Control Statement in Upper Case Letters) for Logical AND

Conditional expression	Generated instruction
α == β &&	CMP(W) α, β BZ $?LTRUE BR ?LFALSE ?LTRUE:
α != β &&	CMP(W) α, β BNZ $?LTRUE BR ?LFALSE ?LTRUE:
α < β &&	CMP(W) α, β BC $?LTRUE BR ?LFALSE ?LTRUE:
α > β &&	CMP(W) α, β BZ $$+4 BNC $?LTRUE BR ?LFALSE ?LTRUE:
α >= β &&	CMP(W) α, β BNC $?LTRUE BR ?LFALSE ?LTRUE:
α <= β &&	CMP(W) α, β BZ $?LTRUE BC $?LTRUE BR ?LFALSE ?LTRUE:
Bit symbol &&	BT Bit symbol, $?LTRUE BR ?LFALSE ?LTRUE:
CY &&	BC $?LTRUE BR ?LFALSE ?LTRUE:
Z &&	BZ $?LTRUE BR ?LFALSE ?LTRUE:
!bit symbol &&	BF Bit symbol, $?LTRUE BR ?LFALSE ?LTRUE:
!CY &&	BNC $?LTRUE BR ?LFALSE ?LTRUE:
!Z &&	BNZ $?LTRUE BR ?LFALSE ?LTRUE:

Logical operations **Logical AND (&&)**

[Use examples]

<1> When the control statement is entered in lower case letters

```
        MOV     A,C             ;if(C==#0 && B>=#0 && B<#80H)(A)
        CMP     A,#0
        BNZ     $?L1
        MOV     A,B
        CMP     A,#0
        BC      $?L1
        MOV     A,B
        CMP     A,#80H
        BNC     $?L1
        CALL    !XXX            ; CALL  !XXX
        BR      ?L2
?L1:                            ;else
        CALL    !YYY            ; CALL  !YYY
?L2:                            ;endif
```

<2> When the control statement is entered in upper case letters

```
        MOV     A,C             ;IF(C==#0 && B>=#0 && B<#80H)(A)
        CMP     A,#0
        BZ      $?L3
        BR      ?L6
?L3:
        MOV     A,B
        CMP     A,#0
        BNC     $?L4
        BR      ?L6
?L4:
        MOV     A,B
        CMP     A,#80H
        BC      $?L5
        BR      ?L6
?L5:
        CALL    !XXX            ; CALL  !XXX
        BR      ?L7
?L6:                            ;ELSE
        CALL    !YYY            ; CALL  !YYY
?L7:                            ;ENDIF
```

(2) Logical OR (||)

[Coding format]

```
Conditional expression 1 [Δ] || [Δ] Conditional expression 2
```

[Function]

The logical OR result of conditional expression 1 and conditional expression 2 is obtained. The result is true when either conditional expression 1 or conditional expression 2 is true and the result is false when both are false. The entered operation is performed when either condition is met.

Instructions for testing are generated first for contents enclosed in parentheses “( )”.

[Generated instructions]

Table 3-8. Generated Instructions for Logical OR

Conditional expression	Generated instruction	
α == β ||	CMP(W) BZ	α, β $?LFALSE
α != β ||	CMP(W) BNZ	α, β $?LFALSE
α < β ||	CMP(W) BC	α, β $?LFALSE
α > β ||	CMP(W) BZ BNC	α, β $$+4 $?LFALSE
α >= β ||	CMP(W) BNC	α, β $?LFALSE
α <= β ||	CMP(W) BZ BC	α, β $?LFALSE $?LFALSE
Bit symbol ||	BT	Bit symbol, $?LFALSE
CY ||	BC	$?LFALSE
Z ||	BZ	$?LFALSE
!bit symbol ||	BF	Bit symbol, $?LFALSE
!CY ||	BNC	$?LFALSE
!Z ||	BNZ	$?LFALSE

Logical operations **Logical OR (||)**

[Use examples]

```
            MOV     A,B             ;if(B==#0 || C>=#0 || D<#80H)(A)
            CMP     A,#0
            BZ      $?L1
            MOV     A,C
            CMP     A,#0
            BNC     $?L1
            MOV     A,D
            CMP     A,#80H
            BNC     $?L2
    ?L1:
            CALL    !XXX            ; CALL  !XXX
            BR      ?L3
    ?L2:                            ;else
            CALL    !YYY            ; CALL  !YYY
    ?L3:                            ;endif
```

[MEMO]

CHAPTER 4 EXPRESSIONS

Expressions are used to perform assignments or arithmetic operations.
The following are examples of expressions

- Assignment statement ····· Assigns the second operand as the first operand
- Count statement ····· Adds or subtracts “1” to the operand value
- Exchange statement ····· Exchanges the values of the first and second operands
- Bit manipulation statement ··· Sets (to 1) or resets (to 0) the value of a operand

Table 4-1. Assignment Statements

Assignment statement		Coding format	Function
(1)	Assign	$\alpha = \beta$	$\alpha \leftarrow \beta$
	Assign (with register specification)	$\alpha = \beta\ (\gamma)$	$(\gamma) \leftarrow \beta,\ \alpha \leftarrow (\gamma)$
	Sequential assign	$\alpha 1 = \cdots = \alpha n = \beta$	$\alpha 1 = \leftarrow \beta, \cdots, \alpha n \leftarrow \beta$
	Sequential assign (with register specification)	$\alpha 1 = \cdots = \alpha n = \beta\ (\gamma)$	$\gamma \leftarrow \beta,\ \alpha 1 \leftarrow \gamma, \cdots, \alpha n \leftarrow \gamma$
(2)	Increment assignment	$\alpha \mathrel{+}= \beta$	$\alpha \leftarrow \alpha + \beta$
	Increment assignment (with register specification)	$\alpha \mathrel{+}= \beta$ (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma + \beta,\ \alpha \leftarrow \gamma$
	Increment assignment (with register specification)	$\alpha \mathrel{+}= \beta$, CY	$\alpha \leftarrow \alpha + \beta$, CY
	Increment assignment (with register specification)	$\alpha \mathrel{+}= \beta$, CY (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma + \beta$, CY, $\alpha \leftarrow \gamma$
(3)	Decrement assignment	$\alpha \mathrel{-}= \beta$	$\alpha \leftarrow \alpha - \beta$
	Decrement assignment (with register specification)	$\alpha \mathrel{-}= \beta$, (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma - \beta,\ \alpha \leftarrow \gamma$
	Decrement assignment (with register specification)	$\alpha \mathrel{-}= \beta$, CY	$\alpha \leftarrow \alpha - \beta$, CY
	Decrement assignment (with register specification)	$\alpha \mathrel{-}= \beta$, CY (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma - \beta$, CY, $\alpha \leftarrow \gamma$
(4)	Logical AND assignment	$\alpha \mathrel{\&}= \beta$	$\alpha \leftarrow \alpha \cap \beta$
	Logical AND assignment (with register specification)	$\alpha \mathrel{\&}= \beta$ (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma \cap \beta,\ \alpha \leftarrow \gamma$
(5)	Logical OR assignment	$\alpha \mathrel{\vert}= \beta$	$\alpha \leftarrow \alpha \cup \beta$
	Logical OR assignment (with register specification)	$\alpha \mathrel{\vert}= \beta$ (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma \cup \beta,\ \alpha \leftarrow \gamma$
(6)	Logical XOR assignment	$\alpha \mathrel{\wedge}= \beta$	$\alpha \leftarrow \alpha \wedge \beta$
	Logical XOR assignment (with register specification)	$\alpha \mathrel{\wedge}= \beta$ (register)	$\gamma \leftarrow \alpha,\ \gamma \leftarrow \gamma \wedge \beta,\ \alpha \leftarrow \gamma$
(7)	Right shift (rotate) assignment	$\alpha >>= \beta$	(α shifted to right of β bit)
	Right shift assignment (with register specification)	$\alpha >>= \beta$ (register)	$\gamma \leftarrow \alpha$, (γ shifted to right of β bit), $\alpha \leftarrow \gamma$
(8)	Left shift assignment	$\alpha <<= \beta$	(α shifted to left of β bit)
	Left shift assignment (with register specification)	$\alpha <<= \beta$ (register)	$\gamma \leftarrow \alpha$, (γ shifted to left of β bit), $\alpha \leftarrow \gamma$

Table 4-2. Count Statements

Count statement		Coding format	Function
(9)	Increment	α ++	$\alpha \leftarrow \alpha + 1$
(10)	Decremen	α –	$\alpha \leftarrow \alpha - 1$

Table 4-3. Exchange Statements

Exchange statement		Coding format	Function
(11)	Exchange	α <-> β	$\alpha \leftarrow \alpha$ <-> β
	Exchange (with register specification)	α <-> β (γ)	$\gamma \leftarrow \alpha$, $\gamma \leftarrow \gamma$ <-> β, $\alpha \leftarrow \gamma$

Table 4-4. Bit Manipulation Statements

Bit manipulation statement		Coding format	Function
(12)	Set bit	$\alpha = 1$	$\alpha \leftarrow 1$
	Set bit (with register specification)	$\alpha = 1$ (CY)	CY $\leftarrow 1$ $\alpha \leftarrow 1$
	Sequential set bit	$\alpha 1 = \cdots = \alpha n = 1$	$\alpha n = \leftarrow 1, \cdots, \alpha 1 \leftarrow 1$
	Sequential set bit (with register specification)	$\alpha 1 = \cdots = \alpha n = 1$ (CY)	CY $\leftarrow 1$, $\alpha n \leftarrow 1, \cdots, \alpha 1 \leftarrow 1$
(13)	Clear bit	$\alpha = 0$	$\alpha \leftarrow 0$
	Clear bit (with register specification)	$\alpha = 0$ (CY)	CY $\leftarrow 0$, $\alpha \leftarrow 0$
	Sequential clear bit	$\alpha 1 = \cdots = \alpha n = 0$	$\alpha n \leftarrow 0, \cdots, \alpha 1 \leftarrow 0$
	Sequential clear bit (with register specification)	$\alpha 1 = \cdots = \alpha n = 0$ (CY)	CY $\leftarrow 0$, $\alpha n \leftarrow 0, \cdots, \alpha 1 \leftarrow 0$

The functions of these expressions are described below.

The generated instructions are shown in the use examples. The input source is shown as comments.

Assignment statements **Assign (=)**

(1) Assign (=)

[Coding format]

```
[Δ] [size specification] [Δ] α 1 [Δ] [= [Δ][size specification] [Δ] α 2 [Δ] ...]
= [Δ][size specification] [Δ] β [Δ][(register specification)]
```

[Function]

<1> When there is no register specification

β values on the right side are sequentially assigned to the left side.

<2> When there is a register specification

β values on the right side are assigned to the specified register or to CY and their contents are sequentially assigned to the left side.

[Description]

α and β are values that can be entered via the MOV or MOVW instruction.

Up to 32 of the assignment operator "=" can be entered in one line. An error occurs when more than 32 are entered. If even one error occurs during sequential assignments, no instructions will be generated.

[Generated instructions]

<1> When α and β are bit symbols

- **When α is CY**

```
    BF      β, ?L1
    SET1    CY
    BR      ?L2
?L1:
    CLR1    CY
?L2:
```

However, sequential assignments cannot be entered.

Assignment statements	Assign (=)

- **When β is CY**

```
    BNC     ?L1
    SET1    α n
    SET1    α n-1
      :
    SET1    α 2
    SET1    α 1
    BR      ?L2
?L1:
    CLR1    α n
    CLR1    α n-1
      :
    CLR1    α 2
    CLR1    α 1
?L2:
```

- **When CY has been specified as the register**

```
    BF      β, ?L1
    SET1    α n
    SET1    α n-1
      :
    SET1    α 2
    SET1    α 1
    BR      ?L2
?L1:
    CLR1    α n
    CLR1    α n-1
      :
    CLR1    α 2
    CLR1    α 1
?L2:
```

Assignment statements **Assign (=)**

<2> When α and β are not bit symbols

- **When there is no register specification**

  MOV α1, β

  MOVW may be generated instead, depending on the operand.

- **When there is no register specification and a sequential assignment has been entered**

  MOV αn, β

  MOV αn-1, β

  :

  MOV α2, β

  MOV α1, β

  MOVW may be generated instead, depending on the operand.

- **When there is a register specification**

  MOV Specified register, α

  MOV α1, specified register

  MOVW may be generated instead, depending on the operand.

- **When there is a register specification and a sequential assignment has been entered**

  MOV Specified register, β

  MOV αn, specified register

  MOV αn-1, specified register

  :

  MOV α2, specified register

  MOV α1, specified register

  MOVW may be generated instead, depending on the operand.

For details of combinations of α and β, see **Table 4-5. Generated Instructions for Assignments**. Depending on the entered statement, αn or β indicates the specified register.

Assignment statements **Assign (=)**

[Use examples]

<1> When there is no register specification

```
        BF      P1.1,$?L1       ;CY=P1.1
        SET1    CY
        BR      ?L2
?L1:
        CLR1    CY
?L2:
        MOV     A,#4H           ;A=#4H
        MOVW    AX,SYMP         ;AX=SYMP
        BNC     $?L3            ;PORT.0=bit1=CY
        SET1    bit1
        SET1    PORT.0
        BR      ?L4
?L3:
        CLR1    bit1
        CLR1    PORT.0
?L4:
        MOV     DAT3,A          ;DAT1=DAT2=DAT3=A
        MOV     DAT2,A
        MOV     DAT1,A
```

<2> When there is a register specification

```
        BF      P1.1,$?L5       ;A.0=P0.2=P1.1(CY)
        SET1    P0.2
        SET1    A.0
        BR      ?L6
?L5:
        CLR1    P0.2
        CLR1    A.0
?L6:
        MOV     A,#4H           ;[DE]=#4H(A)
        MOV     [DE],A
        MOV     A,X             ;DAT1=DAT2=DAT3=X(A)
        MOV     DAT3,A
        MOV     DAT2,A
        MOV     DAT1,A
        MOVW    AX,BC           ;DATA1P=DATA2P=DATA3P=BC(AX)
        MOVW    DATA3P,AX
        MOVW    DATA2P,AX
        MOVW    DATA1P,AX
```

Assignment statements **Assign (=)**

Table 4-5. Generated Instructions for Assignments

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY		*3		*4								*3										
	b	Bit symbol	*3			*5																		
	c	[HL]. β	*3			*5																		
	d	Byte user symbol	*6			*5		*1								*2								*1
	e	Byte data						*1																*1
	f	A				*1	*1		*1	*1											*1	*1	*1	*1
	g	Byte register						*1																*1
	h	R0						*1																*1
	i	R1																						*1
αn	j	sfr						*1																*1
	k	PSW						*1																*1
	l	Word user symbol	*3			*5		*1								*2								
	m	Word data														*2								
	n	AX				*2								*2	*2		*2			*2				*2
	o	Word register														*2								*2
	p	RP0																						*2
	q	sfrp																						
	r	SP														*2								
	s	Direct access symbol						*1																
	t	Indirect access symbol						*1																
	u	[DE]						*1																
	v	Immediate symbol																						

***1**: Generates MOV instruction

***2**: Generates MOVW instruction

***3**: Generates a replacement instruction using a bit branch instruction

***4**: Generates SET1 instruction when “1” has been entered as β. Generates CLR1 instruction when “0” has been entered. Generates MOV when any value other than “0” or “1” has been entered.

***5**: Generates SET1 when “1” has been entered as β. Generates CLR1 when “0” has been entered.

***6**: Generates a replacement instruction using a test bit expression when any value other than “0” or “1” has been entered as α n.

Empty spaces indicate errors.

(2) IncrementAssign (+=)

[Coding format]

```
[Δ] [size specification] [Δ] α 1 [Δ] += [Δ][size specification] [Δ] β [Δ]
                                        [,[Δ] CY] [Δ][(register specification)]
```

[Function]

<1> When there is no register specification

The two operands α and β are added and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

The contents of the specified register are added to β and their result is assigned to the specified register.

The contents of the specified register are assigned to α.

<3> Increment with carry; no register specification

An increment with carry operation is performed using the two operands α and β, and the result is assigned to α.

<4> Increment with carry; with register specification

The contents of α are assigned to the specified register.

An increment with carry operation is performed using the contents of the specified register and β, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> When there is no register specification

The contents of α and β can be entered in ADD and ADDW.

<2> When there is a register specification

The contents of α can be entered in MOV and MOVW.

The contents of β can be entered in ADD and ADDW.

<3> Increment with carry; no register specification

The contents of α and β can be entered in ADDC.

<4> Increment with carry; with register specification

The contents of α can be entered in MOV.

The contents of β can be entered in ADDC.

Assignment statements	IncrementAssign (+=)

[Generated instructions]

<1> When there is no register specification

ADD α, β

ADDW may be generated instead, depending on the operand.

<2> When there is a register specification

MOV Specified register, α

ADD Specified register, β

MOV α, specified register

ADDW may be generated instead, depending on the operand.

<3> Increment with carry; no register specification

ADDC α, β

<4> Increment with carry; with register specification

MOV Specified register, α

ADDC Specified register, β

MOV α, specified register

For details of combinations of α and β, see **Table 4-6. Generated Instructions for Increment Assignments**. Depending on the entered statement, α indicates the specified register.

Assignment statements **IncrementAssign (+=)**

[Use examples]

<1> When there is no register specification

```
ADD         A, #0C0H      ;A+=#0C0H
ADDW        Ax, #0C00H    ;Ax+=#0C00H
```

<2> When there is a register specification

```
MOV         A, !ABC       ;!ABC+=#0FCH(A)
ADD         A, #0FCH
MOV         !ABC, A
MOVW        AX, HL        ;HL+=#0FFFH(AX)
ADDW        AX, #0FFFH
MOVW        HL, AX
```

<3> Increment with carry; no register specification

```
ADDC        A, #50H       ;A+=#50H,CY
```

<4> Increment with carry; with register specification

```
MOV         A, PSW        ;PSW+=#50H,CY(A)
ADDC        A, #50H
MOV         PSW, A
```

Assignment statements **IncrementAssign (+=)**

Table 4-6. Generated Instructions for Increment Assignments

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY																						
	b	Bit symbol																						
	c	[HL].β																						
	d	Byte user symbol																						*1
	e	Byte data																						*1
	f	A				*1	*1		*1	*1	*1			*1							*1	*1		*1
	g	Byte register																						
	h	R0																						
	i	R1																						
α n	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX																						*2
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1**: Generates ADD instruction. For increment with carry, ADDC instruction is generated.

***2**: Generates ADDW instruction.

Empty spaces indicate errors.

(3) DecrementAssign (–=)

[Coding format]

```
[Δ] [size specification] [Δ] α 1 [Δ] –= [Δ][size specification] [Δ] β [Δ]
                                        [,[Δ] CY] [Δ][(register specification)]
```

[Function]

<1> When there is no register specification

β is subtracted from α and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

β is subtracted from the contents of the specified register and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

<3> Decrement with carry; no register specification

A decrement with carry operation is performed using the two operands α and β, and the result is assigned to α.

<4> Decrement with carry; with register specification

The contents of α are assigned to the specified register.

An decrement with carry operation is performed using the contents of the specified register and β, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> When there is no register specification

The contents of α and β can be entered in SUB and SUBW.

<2> When there is a register specification

The contents of α can be entered in MOV and MOVW.

The contents of β can be entered in SUB and SUBW.

<3> Decrement with carry; no register specification

The contents of α and β can be entered in SUBC.

<4> Decrement with carry; with register specification

The contents of α can be entered in MOV.

The contents of β can be entered in SUBC.

Assignment statements **DecrementAssign (–=)**

[Generated instructions]

<1> When there is no register specification

The following instruction is generated.

SUB α, β

SUBW may be generated instead, depending on the operand.

<2> When there is a register specification

The following instruction is generated.

MOV Specified register, α

SUB Specified register, β

MOV α, specified register

SUBW may be generated instead, depending on the operand.

<3> Decrement with carry; no register specification

The following instruction is generated.

SUBC α, β

<4> Decrement with carry; with register specification

The following instruction is generated.

MOV Specified register, α

SUBC Specified register, β

MOV α, specified register

For details of combinations of α and β, see **Table 4-7. Generated Instructions for Decrement Assignments**. Depending on the entered statement, α indicates the specified register.

Assignment statements **DecrementAssign (–=)**

[Use examples]

<1> When there is no register specification

```
SUB         A, #0C0H      ;A–=#0C0H
SUBW        AX, #0C00H    ;AX–=#0C00H
```

<2> When there is a register specification

```
MOV         A, !ABC       ;!ABC–=#0FCH(A)
SUB         A, #0FCH
MOV         !ABC, A
MOVW        AX, HL        ;HL–=#0FFFH(AX)
SUBW        AX, #0FFFH
MOVW        HL, AX
```

<3> Decrement with carry; no register specification

```
SUBC        A, #50H       ;A–=#50H,CY
```

<4> Decrement with carry; with register specification

```
MOV         A, PSW        ;PSW–=#50H,CY(A)
SUBC        A, #50H
MOV         PSW, A
```

Table 4-7. Generated Instructions for Decrement Assignments

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY																						
	b	Bit symbol																						
	c	[HL].β																						
	d	Byte user symbol																						*1
	e	Byte data																						*1
	f	A				*1	*1		*1	*1	*1			*1							*1	*1		*1
	g	Byte register																						
	h	R0																						
	i	R1																						
α n	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX																						*2
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1**: Generates SUB instruction. For decrement with carry, SUBC instruction is generated.

***2**: Generates SUBW instruction.

Empty spaces indicate errors.

(4) LogicalANDAssign (-=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] &= [Δ][size specification] [Δ] β [Δ]
                                                [register specification]
```

[Function]

<1> When there is no register specification

The logical AND (α & β) is obtained from the bits in α and β, and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

The logical AND (α & β) is obtained from the bits in the specified register and β, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> Where there is no register specification

The contents of α and β can be entered in AND and BF.

<2> Where there is a register specification

The contents of α can be entered in MOV and BF.

The contents of β can be entered in AND and BF.

[Generated instructions]

<1> When there is no register specification

- **When α is CY**

```
    BNC     ?L1
    BF      β, ?L1
    SET1    CY
    BR      ?L2
?L1:
    CLR1    CY
?L2:
```

- **When α is not CY**

```
AND         α, β
```

<2> When there is a register specification

- **When the specified register is CY**

```
    BF      α, ?L1
    BF      β, ?L1
    SET1    α
    BR      ?L2
?L1:
    CLR1    α
?L2:
```

- **When the specified register is not CY**

```
MOV     Specified register, α
AND     Specified register, β
MOV     α, specified register
```

For details of combinations of α and β, see **Table 4-8. Generated Instructions for Logical AND Assignments**.

[Use examples]

<1> When there is no register specification

```
            BNC     $?L1    ;CY&=P1S.1
            BF      P1S.1,$?L1
            SET1    CY
            BR      ?L2
?L1:
            CLR1    CY
?L2:
            AND     A,#0FFH ;A&=#0FFH
```

<2> When there is a register specification

```
            BF      A.1,$?L3 ;A.1&=PORT3.0(CY)
            BF      PORT3.0,$?L3
            SET1    A.1
            BR      ?L4
?L3:
            CLR1    A.1
?L4:
            MOV     A,[DE]  ;[DE]&=#07H(A)
            AND     A,#07H
            MOV     [DE],A
```

Assignment statements **LogicalANDAssign (&=)**

Table 4-8. Generated Instructions for Logical AND Assignments

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY		***2**		***2**								***2**										
	b	Bit symbol																						
	c	[HL].β																						
	d	Byte user symbol																						***1**
	e	Byte data																						***1**
	f	A				***1**	***1**		***1**	***1**	***1**			***1**							***1**	***1**		***1**
	g	Byte register																						
	h	R0																						
	i	R1																						
α n	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX																						
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1**: Generates AND instruction.

***2**: Generates a replace instruction depending on the bit branch instruction.

Empty spaces indicate errors.

(5) LogicalORAssign (|=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] |= [Δ][size specification] [Δ] β [Δ]
                                                [register specification]
```

[Function]

<1> When there is no register specification

The logical OR ($\alpha \mid \beta$) is obtained from the bits in α and β, and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

The logical OR ($\alpha \mid \beta$) is obtained from the bits in the specified register and β, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> When there is no register specification

The contents of α and β can be entered in OR and BF.

<2> When there is a register specification

The contents of α can be entered in MOV and BF.

The contents of β can be entered in OR and BF.

[Generated instructions]

<1> When there is no register specification

- **When α is CY**

```
    BC      ?L1
    BF      β, ?L2
?L1:
    SET1    CY
    BR      ?L3
?L2:
    CLR1    CY
?L3:
```

- **When α is not CY**

```
OR          α, β
```

<2> When there is a register specification

- When the specified register is CY

```
    BT      α, ?L1
    BF      β, ?L2
?L1:
    SET1    α
    BR      ?L3
?L2:
    CLR1    α
?L3:
```

- **When the specified register is not CY**

```
MOV     Specified register, α
OR      Specified register, β
MOV     α, specified register
```

For details of combinations of α and β, see **Table 4-9. Generated Instructions for Logical OR Assignments**.

Assignment statements **LogicalORAssign (|=)**

[Use examples]

<1> When there is no register specification

```
        BC      $?L1            ;CY|=P1S.1
        BF      P1S.1,$?L2
?L1:
        SET1    CY
        BR      ?L3
?L2:
        CLR1    CY
?L3:
        OR      A,#0FFH         ;A|=#0FFH
```

<2> When there is a register specification

```
        BT      A.1,$?L4        ;A.1|=PORT3.0(CY)
        BF      PORT3.0,$?L5
?L4:
        SET1    A.1
        BR      ?L6
?L5:
        CLR1    A.1
?L6:
        MOV     A,[DE]          ;[DE]|=#07H(A)
        OR      A,#07H
        MOV     [DE],A
```

Assignment statements **LogicalORAssign (|=)**

Table 4-9. Generated Instructions for Logical OR Assignments

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY		***2**		***2**								***2**										
	b	Bit symbol																						
	c	[HL].β																						
	d	Byte user symbol																						***1**
	e	Byte data																						***1**
	f	A				***1**	***1**		***1**	***1**	***1**			***1**							***1**	***1**		***1**
	g	Byte register																						
	h	R0																						
	i	R1																						
α n	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX																						
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1**: Generates OR instruction.

***2**: Generates a replace instruction depending on the bit branch instruction.

Empty spaces indicate errors.

(6) LogicalXORAssign (∧=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] ∧= [Δ][size specification] [Δ] β [Δ]
                                                [register specification]
```

[Function]

<1> When there is no register specification

The logical XOR ($\alpha \wedge \beta$) is obtained from the bits in α and β, and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

The logical XOR ($\alpha \wedge \beta$) is obtained from the bits in the specified register and β, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> When there is no register specification

The contents of α and β can be entered in XOR and BF.

<2> When there is a register specification

The contents of α can be entered in MOV and BT.

The contents of β can be entered in XOR and BF.

[Generated instructions]

<1> When there is no register specification

- **When α is CY**

```
    BNC     ?L1
    BF      β, ?L2
?L1:
    BC      ?L3
    BF      β, ?L3
?L2:
    SET1    CY
    BR      ?L4
?L3:
    CLR1    CY
?L4:
```

- **When α is not CY**

```
XOR         α, β
```

<2> When there is a register specification

- **When the specified register is CY**

```
    BF    α, ?L1
    BF    β, ?L2
?L1:
    BT    α, ?L3
    BF    β, ?L3
?L2:
    SET1  α
    BR    ?L4
?L3:
    CLR1  α
?L4:
```

- **When the specified register is not CY**

```
MOV    Specified register, α
XOR    Specified register, β
MOV    α, specified register
```

For details of combinations of α and β, see **Table 4-10. Generated Instructions for Logical XOR Assignments**.

Assignment statements **LogicalXORAssign (∧=)**

[Use examples]

<1> When there is no register specification

```
        BNC     $?L1            ;CY^=P1S.1
        BF      P1S.1,$?L2
?L1:
        BC      $?L3
        BF      P1S.1,$?L3
?L2:
        SET1    CY
        BR      ?L4
?L3:
        CLR1    CY
?L4:
        XOR     A,#0FFH         ;A^=#0FFH
```

<2> When there is a register specification

```
        BF      A.1,$?L5        ;A.1^=PORT3.0(CY)
        BF      PORT3.0,$?L6
?L5:
        BT      A.1,$?L7
        BF      PORT3.0,$?L7
?L6:
        SET1    A.1
        BR      ?L8
?L7:
        CLR1    A.1
?L8:
        MOV     A,[DE]          ;[DE]^=#07H(A)
        XOR     A,#07H
        MOV     [DE],A
```

Assignment statements **LogicalXORAssign (∧=)**

Table 4-10. Generated Instructions for Logical XOR Assignments

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	a	CY		***2**		***2**								***2**										
	b	Bit symbol																						
	c	[HL].β																						
	d	Byte user symbol																						***1**
	e	Byte data																						***1**
	f	A				***1**	***1**		***1**	***1**	***1**			***1**							***1**	***1**		***1**
	g	Byte register																						
	h	R0																						
	l	R1																						
α n	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX																						
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1**: Generates XOR instruction.

***2**: Generates a replace instruction depending on the bit branch instruction.

Empty spaces indicate errors.

(7) RightShiftAssign (>>=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] >>= [Δ] β [Δ] [register specification]
```

[Function]

<1> When there is no register specification

α is shifted to the right of the β bit, and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

The contents of the specified register are shifted to the right of the β bit, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> When there is no register specification

The contents of α can be entered in A only.

The contents of β can be entered as numerals from 1 to 7.

<2> When there is a register specification

The contents of α can be entered in MOV.

The contents of β can be entered as numerals from 1 to 7.

The specified register can be entered in A only.

[Generated instructions]

<1> When there is no register specification

An AND instruction is generated after a ROR instruction is output β times.

```
ROR     A, 1
:
AND     A, #0FFH SHR β
```

<2> When there is a register specification

```
MOV     A, α
ROR     A, 1
:
AND     A, #0FFH SHR β
MOV     α, A
```

Assignment statements **RightShiftAssign (>>=)**

[Use examples]

<1> When there is no register specification

```
ROR    A,1           ;A>>=4
ROR    A,1
ROR    A,1
ROR    A,1
AND    A,#0FFH SHR 4
```

<2> When there is a register specification

```
MOV    A,CCV         ;CCV>>=4(A)
ROR    A,1
ROR    A,1
ROR    A,1
ROR    A,1
AND    A,#0FFH SHR 4
MOV    CCV,A
```

(8) LeftShiftAssign (<<=)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] <<= [Δ] β [Δ] [register specification]
```

[Function]

<1> When there is no register specification

α is shifted to the left of the β bit, and the result is assigned to α.

<2> When there is a register specification

α is assigned to the specified register.

The contents of the specified register are shifted to the left of the β bit, and the result is assigned to the specified register.

The contents of the specified register are assigned to α.

[Description]

<1> When there is no register specification

The contents of α can be entered in A only.

The contents of β can be entered as numerals from 1 to 7.

<2> When there is a register specification

The contents of α can be entered in MOV.

The contents of β can be entered as numerals from 1 to 7.

The specified register can be entered in A only.

[Generated instructions]

<1> When there is no register specification

An AND instruction is generated after a ROL instruction is output β times.

```
ROL     A, 1
:
AND     A, #LOW(0FFH SHL β )
```

<2> When there is a register specification

```
MOV     A, α
ROL     A, 1
:
AND     A, #LOW(0FFH SHL β )
MOV     α, A
```

Assignment statements **LeftShiftAssign (>>=)**

[Use examples]

<1> When there is no register specification

```
ROL    A,1           ;A<<=4
ROL    A,1
ROL    A,1
ROL    A,1
AND    A,#LOW( 0FFH SHL 4 )
```

<2> When there is a register specification

```
MOV    A,CCV         ;CCV<<=4(A)
ROL    A,1
ROL    A,1
ROL    A,1
ROL    A,1
AND    A,#LOW( 0FFH SHL 4 )
MOV    CCV,A
```

(9) Increment (++)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] ++
```

[Function]

1 is added to the contents of α.

[Description]

The contents of α can be entered in INC or INCW.

[Generated instructions]

INCW α

DECW may be generated depending on the operands.

For details of α, see **Table 4-11. Generated Instructions for Increment**.

[Use examples]

```
INC     H       ;H++
INC     CNT     ;CNT++
INCW    HL      ;HL++
```

Table 4-11. Generated Instructions for Increment

α	a	CY	
	b	Bit symbol	
	c	[HL].β	
	d	Byte user symbol	***1**
	e	Byte data	***1**
	f	A	***1**
	g	Byte register	***1**
	h	R0	***1**
	i	R1	***1**
	j	sfr	
	k	PSW	
	l	Word user symbol	
	m	Word data	
	n	AX	***2**
	o	Word register	***2**
	p	RP0	***2**
	q	sfrp	
	r	SP	
	s	Direct access symbol	
	t	Indirect access symbol	
	u	[DE]	
	v	Immediate symbol	

***1**: Generates INC instruction.
***2**: Generates INCW instruction.
Empty spaces indicate errors.

(10) Decrement (– –)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ]– –
```

[Function]

1 is subtracted from the contents of α.

[Description]

The contents of α can be entered in DEC or DECW.

[Generated instructions]

DEC α

DECW may be generated depending on the operands.

For details of α, see **Table 4-12. Generated Instructions for Decrement**.

[Use examples]

```
DEC     H       ;H– –
DEC     CNT     ;CNT– –
DECW    HL      ;HL– –
```

Table 4-12. Generated Instructions for Decrement

α	a	CY	
	b	Bit symbol	
	c	[HL].β	
	d	Byte user symbol	***1**
	e	Byte data	***1**
	f	A	***1**
	g	Byte register	***1**
	h	R0	***1**
	i	R1	***1**
	j	sfr	
	k	PSW	
	l	Word user symbol	
	m	Word data	
	n	AX	***2**
	o	Word register	***2**
	p	RP0	***2**
	q	sfrp	
	r	SP	
	s	Direct access symbol	
	t	Indirect access symbol	
	u	[DE]	
	v	Immediate symbol	

***1**: Generates DEC instruction.
***2**: Generates DECW instruction.
Empty spaces indicate errors.

(11) Exchange (<->)

[Coding format]

```
[Δ] [size specification] [Δ] α [Δ] <-> [Δ][size specification] [Δ] β [Δ]
                                          [(register specification)]
```

[Function]

<1> When there is no register specification

The contents of α and β are exchanged.

<2> When there is a register specification

The contents of α are assigned to the specified register.

The contents of the specified register are exchanged with the contents of β.

The contents of the specified register are assigned to α.

[Description]

<1> Where there is no register specification

The contents of α and β can be entered in XCH or XCHW.

<2> When there is a register specification

The contents of α can be entered in MOV and MOVW.

The contents of β can be entered in XCH and XCHW.

[Generated instructions]

<1> When there is no register specification

XCH α, β

XCHW may be generated depending on the operands.

<2> When there is a register specification

MOV Specified register, α

XCH Specified register, β

MOV α, specified register

XCHW may be generated depending on the operands.

For details of combinations of α and β, see **Table 4-13. Generated Instructions for Exchange**.

α indicates the specified register.

Exchange statements **Exchange (<->)**

[Use examples]

<1> When there is no register specification

```
XCH         A,B         ;A<->B
XCHW        AX,BC       ;AX<->BC
```

<2> When there is a register specification

```
MOV         A,DATA      ;DATA<->B(A)
XCH         A,B
MOV         DATA,A
MOVW        AX,DE       ;DE<->BC(AX)
XCHW        AX,BC
MOVW        DE,AX
```

Exchange statements **Exchange (<->)**

Table 4-13. Generated Instructions for Exchange

			β																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
α	a	CY																						
	b	Bit symbol																						
	c	[HL].β																						
	d	Byte user symbol																						
	e	Byte data																						
	f	A			*1	*1		*1			*1		*1									*1	*1	
	g	Byte register																						
	h	R0																						
	l	R1																						
	j	sfr																						
	k	PSW																						
	l	Word user symbol																						
	m	Word data																						
	n	AX															*2							
	o	Word register																						
	p	RP0																						
	q	sfrp																						
	r	SP																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	[DE]																						
	v	Immediate symbol																						

***1**: Generates XCH instructions.

***2**: Generates XCHW instructions.

Empty spaces indicate errors.

(13) Set bit (=)

[Coding format]

```
[Δ] α1 [Δ] [=[Δ] α2 [Δ]…] = [Δ] 1 [Δ] [(CY specification)]
```
Enter a “1” at the end of the right side.

[Function]

<1> When there is no CY specification

αn is set (to a value of “1”).

<2> When there is a CY specification

CY and αn are set (to a value of “1”).

[Description]

The contents of αn can be entered in a SET1 instruction.

Up to 32 of the assignment operator “=” can be entered in one line. An error occurs when more than 32 are entered. If even one error occurs during sequential assignments, no instructions will be generated.

[Generated instructions]

<1> When there is no CY specification

SET1 α1

<2> When there is no CY specification in sequential assignments

SET1 αn
SET1 αn-1
:
SET1 α2
SET1 α1

<3> When there is a CY specification

SET1 CY
SET1 α1

Bit manipulation statements **Set bit (=)**

<4> When there is a CY specification in sequential assignments

SET1 CY
SET1 αn
SET1 αn-1
:
SET1 α2
SET1 α1

For details, see **Table 4-14. Generated Instructions for Set Bit**

[Use examples]

<1> When there is no CY specification

```
SET1   A.3          ;A.3=1
SET1   CY           ;CY=1
SET1   BIT3         ;BIT1=BIT2=BIT3=1
SET1   BIT2
SET1   BIT1
```

<2> When there is a CY specification

```
SET1   CY           ;A.5=1(CY)
SET1   A.5
SET1   CY           ;BIT1=BIT2=BIT3=1(CY)
SET1   BIT3
SET1   BIT2
SET1   BIT1
```

Bit manipulation statements **Set bit (=)**

Table 4-14. Generated Instructions for Set Bit

α	a	CY	***1**
	b	Bit symbol	***1**
	c	[HL].β	***1**
	d	Byte user symbol	***1**
	e	Byte data	
	f	A	
	g	Byte register	
	h	R0	
	i	R1	
	j	sfr	
	k	PSW	
	l	Word user symbol	***1**
	m	Word data	
	n	AX	
	o	Word register	
	p	RP0	
	q	sfrp	
	r	SP	
	s	Direct access symbol	
	t	Indirect access symbol	
	u	[DE]	
	v	Immediate symbol	

***1**: Generates SET1 instruction.
Empty spaces indicate errors.

(14) Clear bit (=)

[Coding format]

```
[Δ] α 1 [=[Δ] α 2 [Δ]···] = [Δ] 0 [Δ] [(CY specification)]
Enter a "0" at the end of the right side.
```

[Function]

<1> When there is no CY specification

αn is cleared (to a value of "0").

<2> When there is a CY specification

CY and αn are cleared (to a value of "0").

[Description]

The contents of αn can be entered in a CLR1 instruction.

Up to 32 of the assignment operator "=" can be entered in one line. An error occurs when more than 32 are entered. If even one error occurs during sequential assignments, no instructions will be generated.

[Generated instructions]

<1> When there is no CY specification

CLR1 α1

<2> When there is no CY specification in sequential assignments

CLR1 αn
CLR1 αn-1
:
CLR1 α2
CLR1 α 1

<3> When there is a CY specification

CLR1 CY
CLR1 α1

Bit manipulation statements **Clear bit (=)**

<4> When there is a CY specification in sequential assignments

CLR1 CY
CLR1 αn
CLR1 αn-1
:
CLR1 α2
CLR1 α1

For details, see **Table 4-15. Generated Instructions for Clear Bit**

[Use examples]

<1> When there is no CY specification

```
CLR1   A.3          ;A.3=0
CLR1   CY           ;CY=0
CLR1   BIT3         ;BIT1=BIT2=BIT3=0
CLR1   BIT2
CLR1   BIT1
```

<2> When there is a CY specification

```
CLR1   CY           ;A.5=0(CY)
CLR1   A.5
CLR1   CY           ;BIT1=BIT2=BIT3=0(CY)
CLR1   BIT3
CLR1   BIT2
CLR1   BIT1
```

Table 4-15. Generated Instructions for Clear Bit

α	a	CY	***1**
	b	Bit symbol	***1**
	c	[HL].β	***1**
	d	Byte user symbol	***1**
	e	Byte data	
	f	A	
	g	Byte register	
	h	R0	
	i	R1	
	j	sfr	
	k	PSW	
	l	Word user symbol	***1**
	m	Word data	
	n	AX	
	o	Word register	
	p	RP0	
	q	sfrp	
	r	SP	
	s	Direct access symbol	
	t	Indirect access symbol	
	u	[DE]	
	v	Immediate symbol	

***1**: Generates CLR1 instruction.
Empty spaces indicate errors.

[MEMO]

CHAPTER 5 DIRECTIVES

This chapter describes directives. In this case, “directives” means various directives that the ST78K0S requires to execute a series of processes.

5.1 Overview of Directives

Directives are entered into source programs as various directives that the ST78K0S requires to execute a series of processes.

The use of directives can make source program coding easier.

Directives are not output in output files.

5.2 Directive Functions

The various types of directives are listed in **Table 5-1. List of Directives**.

Table 5-1. List of Directives

Type of directive	Directive name
Symbol definition directive	#define
Conditional processing directive	#ifdef : #else : #endif
Include directive	#include
CALLT replacement directive	#defcallt : #endcallt

The directives’ functions are described below.

(1) Symbol definition directive (#define)

[Coding format]

```
[Δ] # [Δ] define Δ symbol Δ character string
```

[Function]

This directive replaces the specified character string with a symbol that has been entered in the source program.

[Description]

<1> The "#" character must always be entered at the start of the symbol, except when starting with a white space or a horizontal tab.

<2> Symbols start with an English letter and are composed of English alphabet letters and numerals, and their valid length is 31 characters by default or 8 characters if the "NS" option has been specified. When a valid length of 8 characters has been specified, only the first 8 characters are read in symbol names having 9 or more characters, and all subsequent characters are ignored. When a valid length of 31 characters has been specified, only the first 31 characters are read in symbol names having 32 or more characters, and all subsequent characters are ignored.

<3> Character strings are defined as strings of characters from among the characters in the set listed in **"2.2 (1) Character set"**. They cannot include white spaces or quotation marks. Any character strings that contain white spaces or quotation marks will be ignored as processing continues.

<4> This directive is useful when coding easy-to-read symbols, such as numerical values.

<5> Reserved words cannot be entered as symbols.

<6> Reserved words can be entered as character strings.

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

<8> Character strings that have been converted to secondary source files are output. The #define statement is not output.

<9> If a converted character string has already been defined by another #define statement, it can be reconverted up to 31 times. An error message is output during the 32nd conversion, and the definition is ignored during subsequent conversions.

<10> This directive can be entered anywhere in the source code.

<11> A warning message is output when two or more symbols specifying option D are entered, and the #define statement is valid.

[Use examples]

<Input source program>

```
#define TRUE     1
   X = #0
   CALL   !xxx
   if( X == #TRUE )
          A = #0C5H
   endif
```

<Output source program>

```
             MOV     X,#0    ; X = #0
             CALL    !xxx    ; CALL  !xxx
             MOV     A,X     ; if( X == #1 )(A)
             CMP     A,#1
             BNZ     $?L1
             MOV     B,#0C5H ;  B = #0C5H
?L1:                         ; endif
```

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

[Coding format]

```
[Δ] # [Δ] ifdef Δ symbol
      text 1
[Δ] # [Δ] else
      text 2
[Δ] # [Δ] endif
```

[Function]

This directive performs conditional processing.

<1> When the symbol has not been defined

If #else has been entered, text 1 is skipped and text 2 becomes a processing object.

<2> When the symbol has been defined

If #else has been entered, text 1 becomes a processing object and text 2 is skipped.

[Description]

<1> The “#” character must always be entered at the start of the symbol, except when starting with a white space or a horizontal tab.

<2> Symbols start with an English letter and are composed of English alphabet letters and numerals, and their valid length is 31 characters by default or 8 characters if the “NS” option has been specified.

<3> Symbols are defined by a previously entered #define statement or by specifying the “-D” option at startup.

<4> This directive can be nested in up to eight levels.

<5> #else can be omitted.

#IFDEF/#ELSE/#ENDIF **#ifdef/#else/#endif** **#IFDEF/#ELSE/#ENDIF**

[Use examples]

<Input source program>

```
#ifdef SYM
   A = #00H
#else
   A = #0FFH
#endif
```

<1> When the following has been entered on the command line (and the symbol has been defined)

```
A>st78k0s -cp9014 sample.st -dSYM
```

<Output source program>

```
MOV      A,#00H  ;       A = #00H
```

<2> When the following has been entered on the command line (and the symbol has not been defined)

```
A>st780s -cp9014 sample.st
```

<Output source program>

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

(3) Include directive (#include)

[Coding format]

```
[Δ] # [Δ] include Δ "file name"
```

[Function]

This line is replaced by the specified file name and becomes a processing object as the ST78K0S source program.

[Description]

<1> The "#" character must always be entered at the start of the symbol, except when starting with a white space or a horizontal tab.

<2> This directive can be entered in any line in the source program.

<3> An include directive cannot be entered in an include file. In other words, nesting of include directives is not allowed.

<4> Input source file names specified at startup, output file names, and error file names cannot be specified as the file name in this directive.

<5> Drive and directory names can be entered before file names. If no drive or directory is entered, processing assumes that the include file belongs to the current drive and current directory.

<6> The -I option can be used to specify a drive and directory for the include file when the ST78K0S is activated.

#INCLUDE #include #INCLUDE

[Use examples]

<Input source program>

```
#include "sample.inc"
   A = SYM1
   B = SYM2
```

<Input include program>

```
#define   SYM1    #08H
#define   SYM2    #0AH
```

<Output source program>

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

(4) CALLT replacement directive (#defcallt)

[Coding format]

```
[Δ] # [Δ] defcallt Δ CALLT table label
[Δ] CALLΔ     ! label
[Δ] # [Δ] endcallt
```

[Function]

The CALL instruction for a registered label is replaced by a CALLT instruction and is output to a secondary file.

[Description]

<1> This directive defines labels that can be registered to the CALLT table, as opposed to the CALL instructions that are entered into the source program. All of the CALL instructions for these defined labels are replaced by CALLT labels.

<2> This directive can be defined up to 32 times. An error message is output during the 33rd definition, and the definition is ignored as processing continues.

<3> If the same pattern is defined twice, an error message is output and the second definition is ignored as processing continues.

[Use examples]

<Input source program>

```
#defcallt        @ABC
   CALL          !abc
#endcallt
R0 = #0
call             !abc
call             !label
```

<Output source program>

```
MOV              R0,#0          ;R0 = #0
CALLT            [@ABC]         ;call   !abc
call             !label         ;call   !label
```

CHAPTER 6 CONTROL INSTRUCTIONS

This chapter describes structured assembler control instructions. Control instructions provide detailed instructions for the structured assembler's operations.

6.1 Overview of Control Instructions

Control instructions, which are entered into the source program, set various directives that the ST78K0S requires to execute a series of processes.

Entering control instructions saves the time that would otherwise be required for specifying options when activating a program.

6.2 Assembler Control Instructions

First, it must be determined whether or not each assembler control instruction can be entered in a module header.

If there is an assembler control instruction that cannot be entered in a module header, subsequent processing proceeds as the module body. If an assembler control instruction that can only be entered in a module header is instead entered in a module body, an error message is output and processing is aborted.

This preprocessor does not confirm the accuracy of parameter specifications except for processor type specification control instructions ($PROCESSOR, $PC), symbol name length control instructions ($SYMLEN, $NOSYMLEN), and kanji code specification control instructions ($KANJICODE). For description of the coding format for other control instructions, see the "**RA78K0S Series Assembler Package User's Manual Assembly Language** ".

The following tables list control instructions that can be entered only in module headers and control instructions that are recognized as the module body.

Table 6-1. Control Instructions that Can Be Entered Only in Module Headers

Control instruction
[Δ] $ [Δ] PROCESSOR [Δ] ([Δ] model name [Δ])
[Δ] $ [Δ] PC ([Δ] model name [Δ])
[Δ] $ [Δ] DEBUG
[Δ] $ [Δ] DG
[Δ] $ [Δ] NODEBAG
[Δ] $ [Δ] NODG
[Δ] $ [Δ] DEBUGA
[Δ] $ [Δ] NODEBAGA
[Δ] $ [Δ] XREF
[Δ] $ [Δ] XR
[Δ] $ [Δ] NOXREF
[Δ] $ [Δ] NOXR
[Δ] $ [Δ] TITLE [Δ] ([Δ] ‘title string’ [Δ])
[Δ] $ [Δ] TT [Δ] ([Δ] ‘title string’ [Δ])
[Δ] $ [Δ] SYMLEN
[Δ] $ [Δ] NOSYMLEN
[Δ] $ [Δ] CAP
[Δ] $ [Δ] NOCAP
[Δ] $ [Δ] SYMLIST
[Δ] $ [Δ] NOSYMLIST
[Δ] $ [Δ] FORMFEED
[Δ] $ [Δ] NOFORMFEED
[Δ] $ [Δ] WIDTH [Δ] ([Δ] constant [Δ])
[Δ] $ [Δ] LENGTH [Δ] ([Δ] constant [Δ])
[Δ] $ [Δ] TAB [Δ] ([Δ] constant [Δ])
[Δ] $ [Δ] KANJICODE Δ kanji code

Table 6-2. Control Instructions that Are Recognized as the Module Body

Control instruction
[Δ] $ [Δ] INCULUDE [Δ] ([Δ] file name [Δ])
[Δ] $ [Δ] IC ([Δ] file name [Δ])
[Δ] $ [Δ] EJECT
[Δ] $ [Δ] EJ
[Δ] $ [Δ] LIST
[Δ] $ [Δ] LI
[Δ] $ [Δ] NOLIST
[Δ] $ [Δ] NOLI
[Δ] $ [Δ] GEN
[Δ] $ [Δ] NOGEN
[Δ] $ [Δ] COND
[Δ] $ [Δ] NOCOND
[Δ] $ [Δ] SUBTITLE [Δ] ([Δ] ‘character string’ [Δ])
[Δ] $ [Δ] ST [Δ] ([Δ] ‘character string’ [Δ])
[Δ] $ [Δ] SET [Δ] ([Δ] switch name [[Δ] : [Δ] switch name… [Δ])
[Δ] $ [Δ] RESET [Δ] ([Δ] switch name [[Δ] : [Δ] switch name… [Δ])
[Δ] $ [Δ] IF [Δ] ([Δ] switch name [[Δ] : [Δ] switch name… [Δ])
[Δ] $ [Δ] _IF Δ conditional expression
[Δ] $ [Δ] ELSEIF [Δ] ([Δ] switch name [[Δ] : [Δ] switch name… [Δ])
[Δ] $ [Δ] _ELESEIF Δ conditional expression
[Δ] $ [Δ] SET [Δ] ([Δ] switch name [[Δ] : [Δ] switch name… [Δ])
[Δ] $ [Δ] ELSE
[Δ] $ [Δ] ENDIF

6.3 Control Instruction Functions

The various functions of control instructions are listed in **Table 6-3. Control Instruction List** below.

Table 6-3. Control Instruction List

Type of control instruction	Control instruction
Processor type specification instruction	$PROCESSOR
Symbol name length control instructions	$SYMLEN/$NOSYMLEN
Kanji code specification control instructions	$KANJICODE

The functions of these three types of control instructions are described below.

(1) Processor type specification instruction ($PROCESSOR)

[Coding format]

```
[Δ] $ [Δ] PROCESSOR [Δ] ( [Δ] model name [Δ] )
[Δ] $ [Δ] PC [Δ] ( [Δ] model name [Δ] )                    ; Abbreviated form
```

[Function]

This control instruction specifies the model in the source module that is the object for assembly.

[Description]

<1> Although this control instruction specifies the model that is the object for assembly by the assembler, it can also be used to specify the model that is the object for the structured assembler.

<2> If the specified model differs from that specified via the “-C” option, the model specified via the “-C” option takes priority. When such a conflict arises, a warning message is output. The “$” in the input source file's control instruction is replaced by a “;” in the secondary source file that is output, and the model specified via an option is output as the processor model specification control instruction. No message is output if the same model name is specified by the “-C” option. If there is no specification via the “-C” option, the specification must be entered at the start of the source module (not including spaces or comments).

<3> An error occurs when this control instruction is entered more than once.

<4> An error occurs if neither this control instruction nor the “-C” option is used to specified a model name.

<5> An error occurs if this control instruction is entered anywhere other than in the module header.

[Code example]

```
$PROCESSOR (P9014)
$PC (P9014)
```

(2) Symbol name length control instructions

[Coding format]

```
[Δ] $ [Δ] SYMLEN
[Δ] $ [Δ] NOSYMLEN
```

[Function]

The SYMLEN control instruction sets a valid length of up to 31 characters for symbol names defined via #define, symbol names accessed via #ifdef, and user symbols.

The NOSYMLEN control instruction sets a valid length of up to 8 characters for symbol names defined via #define, symbol names accessed via #ifdef, and user symbols.

[Description]

<1> This control instruction can be entered in the module header section of an input source file.

<2> An error occurs if this control instruction is entered anywhere other than in the module header.

<3> If this control instruction is entered more than once, the most recent one takes priority.

<4> The symbol name length control instructions can also be specified via the command line options “-S” and “-NS”. Those options take priority over this control instruction.

<5> The default interpretation is $SYMLEN.

<6> If the “-S” option has been specified and $NOSYMLEN has been entered in the input source file, comments will be replaced and $SYMLEN will be output to a secondary source file.
If the “-NS” option has been specified and $SYMLEN has been entered in the input source file, comments will be replaced and $NOSYMLEN will be output to a secondary source file.

[Code example]

```
$SYMLEN
$NOSYMLEN
```

(3) Kanji code specification control instruction ($KANJICODE)

[Coding format]

```
[Δ] $ [Δ] KANJICODE Δ kanji code
```

[Function]

The kanji codes used in comments are interpreted as follows.

Table 6-4. Interpretation of Kanji Code

Kanji code	Interpretation
SJIS	Interpreted as SHIFT-JIS code
EUC	Interpreted as EUC code
NONE	Not interpreted as kanji code

[Description]

<1> This control instruction can be entered in the module header section of an input source file.

<2> An error occurs if this control instruction is entered anywhere other than in the module header.

<3> If this control instruction is entered more than once, the most recent one takes priority.

<4> This preprocessor outputs the specified control instruction to a secondary source file.

SJIS : $KANJICODE SJIS

EUC : $KANJICODE EUC

NONE : $KANJICODE NONE

If the same control instruction is entered in a secondary source file, the control instruction is not output. However, error checking is performed.

<5> For a list of priority ranking among kanji code specifications, see **"1.3.3 Environment variables"**.

[Code example]

```
$KANJICODE SJIS
```

[MEMO]

APPENDIX A SYNTAX LISTS

Table A-1. Control Statements

Control statement	Coding format	Page

if statement	if (conditional expression 1) [(register name)]
	  if block
	elseif (conditional expression 2) [(register name)]
	  elseif block
	else
	  else block
	endif
		P.21

switch statement	switch (symbol) [(register name)]
	  case constant 1:
	  case1 block
	  case constant 2
	  case2 block
	  :
	  case constant N
	  caseN block
	  default:
	  default block
	ends
		P.27

for statement	for (expression; conditional expression; expression) [(register name)] Instruction group next	P.31
while statement	while (conditional expression) [(register name)] Instruction group endw	P.34
until statement	repeat Instruction group until (conditional expression) [(register name)]	P.38
break statement	break	P.41
continue statement	continue	P.42
goto statement	goto label	P.43

if_bit statement	if_bit (conditional expression 1) [(register name)]
	  if_bit block
	elseif_bit (conditional expression 2) [(register name)]
	  elseif_bit block
	else
	  else block
	endif
		P.24

while_bit statement	while_bit (conditional expression) [(register name)] Instruction group endw	P.36
until_bit statement	repeat Instruction group until_bit (conditional expression) [(register name)]	P.40

Table A-2. Conditional Expressions

Conditional expression	Coding format	Function	Page
Equal	α == β	True when $\alpha = \beta$, false when $\alpha \neq \beta$	P.47
NotEqual	α != β	True when $\alpha \neq \beta$, false when $\alpha = \beta$	P.50
LessThan	$\alpha < \beta$	True when $\alpha < \beta$, false when $\alpha >= \beta$	P.53
GreaterThan	$\alpha > \beta$	True when $\alpha > \beta$, false when $\alpha <= \beta$	P.56
GreaterEqual	$\alpha >= \beta$	True when $\alpha >= \beta$, false when $\alpha < \beta$	P.59
LessEqual	$\alpha <= \beta$	True when $\alpha <= \beta$, false when $\alpha > \beta$	P.62
FOREVER	forever	Sets endless loop for loop statement	P.65
Positive logic (bit)	Bit symbol	True when value of specified bit symbol is 1	P.68
Negative logic (bit)	!bit symbol	True when value of specified bit symbol is 0	P.71
Logical AND	Conditional expression 1 && conditional expression 2	True when both conditional expression 1 and conditional expression 2 are true	P.75
Logical OR	Conditional expression 1 || conditional expression 2	True when either conditional expression 1 or conditional expression 2 is true	P.78

Table A-3. Expressions (1/2)

Expression	Coding format	Function	Page
Assign	$\alpha = \beta$	$\alpha \leftarrow \beta$	P.83
Assign (with register specification)	$\alpha = \beta\ (\gamma)$	$(\gamma) \leftarrow \beta\ \ \alpha \leftarrow (\gamma)$	
Sequential assign	$\alpha 1 = \cdots = \alpha n = \beta$	$\alpha 1 \leftarrow \beta, \cdots, \alpha n \leftarrow \beta$	
Sequential assign (with register specification)	$\alpha 1 = \cdots = \alpha n = \beta\ (\gamma)$	$\gamma \leftarrow \beta, \alpha 1 \leftarrow \gamma, \cdots, \alpha n \leftarrow \gamma$	
Increment assignment	α += β	$\alpha \leftarrow \alpha + \beta$	P.88
Increment assignment (with register specification)	α += β (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma + \beta, \alpha \leftarrow \gamma$	
Increment assignment (with register specification)	α += β, CY	$\alpha \leftarrow \alpha + \beta$, CY	
Increment assignment (with register specification)	α += β, CY (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma + \beta$, CY, $\alpha \leftarrow \gamma$	
Decrement assignment	α –= β	$\alpha \leftarrow \alpha - \beta$	P.92
Decrement assignment (with register specification)	α –= β (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma - \beta, \alpha \leftarrow \gamma$	
Decrement assignment (with register specification)	α –= β, CY	$\alpha \leftarrow \alpha - \beta$, CY	
Decrement assignment (with register specification)	α –= β, CY (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma - \beta$, CY, $\alpha \leftarrow \gamma$	
Logical AND assignment	α &= β	$\alpha \leftarrow \alpha \cap \beta$	P.96
Logical AND assignment (with register specification)	α &= β (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \cap \beta, \alpha \leftarrow \gamma$	

Table A-3. Expressions (2/2)

Expression	Coding format	Function	Page
Logical OR assignment	α |= β	$\alpha \leftarrow \alpha \cup \beta$	P.99
Logical OR assignment (with register specification)	α |= β (Register)	$\gamma \leftarrow \alpha$, $\gamma \leftarrow \gamma \cup \beta$, $\alpha \leftarrow \gamma$	
Logical XOR assignment	α ^= β	$\alpha \leftarrow \alpha \hat{} \beta$	P.103
Logical XOR assignment (with register specification)	α ^= β (Register)	$\gamma \leftarrow \alpha$, $\gamma \leftarrow \gamma \hat{} \beta$, $\alpha \leftarrow \gamma$	
Right shift (rotate) assignment	α >>= β	(α shifted to right of β bit)	P.107
Right shift assignment (with register specification)	α >>= β (Register)	$\gamma \leftarrow \alpha$,(γ shifted to right of β bit), $\alpha \leftarrow \gamma$	
Left shift assignment	α <<= β	(α shifted to left of β bit)	P.109
Left shift assignment (with register specification)	α <<= β (Register)	$\gamma \leftarrow \alpha$, (γ shifted to left of β bit), $\alpha \leftarrow \gamma$	
Increment	α ++	$\alpha \leftarrow \alpha + 1$	P.111
Decrement	α – –	$\alpha \leftarrow \alpha - 1$	P.113
Exchange	α <–>= β	$\alpha \leftarrow \alpha$ <–>= β	P.115
Exchange (with register specification)	α <–>= β (γ)	$\gamma \leftarrow \alpha$, $\gamma \leftarrow \gamma$ <–> β, $\alpha \leftarrow \gamma$	
Set bit	α = 1	$\alpha \leftarrow 1$	P.118
Set bit (with register specification)	α = 1 (CY)	CY $\leftarrow$ 1, $\alpha \leftarrow 1$	
Sequential set bit	α 1 = ⋯ =α n = 1	α n $\leftarrow$ 1, ⋯ , α 1 $\leftarrow$ 1	
Sequential set bit (with register specification)	α 1 = ⋯ =α n = 1 (CY)	CY $\leftarrow$ 1, α n $\leftarrow$ 1, ⋯ , α 1 $\leftarrow$ 1	
Clear bit	α = 0	$\alpha \leftarrow 0$	P.121
Clear bit (with register specification)	α = 0 (CY)	CY $\leftarrow$ 0, $\alpha \leftarrow 0$	
Sequential clear bit	α 1 = ⋯ =α n = 0	α n $\leftarrow$ 0, ⋯ , α 1 $\leftarrow$ 0	
Sequential clear bit (with register specification)	α 1 = ⋯ =α n = 0 (CY)	CY $\leftarrow$ 0, α n $\leftarrow$ 0, ⋯ , α 1 $\leftarrow$ 0	

Table A-4. Directives

Directive	Coding format	Page
#define	#define symbol character string	P.126

#ifdef	#ifdef symbol
	  text 1
	#else
	  text 2
	#endif
		P.128

#include	#include “file name”	P.130
#defcallt	#defcallt CALLT table label CALL label #endcallt	P.132

[MEMO]

APPENDIX B LISTS OF GENERATED INSTRUCTIONS

Table B-1. Generated Instructions for Comparison Expressions (1/3)

Comparison expression	Generated instruction	Control statement condition	Page
$\alpha == \beta$	CMP(W) α, β BNZ $?LFALSE	lower case letters	P.47
	CMP(W) α, β BZ $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha == \beta\ (\gamma)$	MOV(W) γ, α CMP(W) γ, β BNZ $?LFALSE	lower case letters	
	MOV(W) γ, α CMP(W) γ, β BZ $?LTRUE BR LFALSE ?LTRUE:	upper case letters	
$\alpha! = \beta$	CMP(W) α, β BZ $?LFALSE	lower case letters	P.50
	CMP(W) α, β BNZ $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha! = \beta\ (\gamma)$	MOV(W) γ, α CMP(W) γ, β BZ $?LFALSE	lower case letters	
	MOV(W) γ, α CMP(W) γ, β BNZ $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha < \beta$	CMP(W) α, β BNC $?LFALSE	lower case letters	P.53
	CMP(W) α, β BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha < \beta\ (\gamma)$	MOV(W) γ, α CMP(W) γ, β BNC $?LFALSE	lower case letters	
	MOV(W) γ, α CMP(W) γ, β BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	

Table B-1. Generated Instructions for Comparison Expressions (2/3)

Comparison expression	Generated instruction	Control statement condition	Page
$\alpha > \beta$	CMP(W) α, β BZ $?LFALSE BC $?LFALSE	lower case letters	P.56
	CMP(W) α, β BZ $$+4 BNC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha > \beta$ (γ)	MOV(W) specified register, α CMP(W) specified register, β BZ $?LFALSE BC $?LFALSE	lower case letters	
	MOV(W) specified register, α CMP(W) specified register, β BZ $$+4 BNC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha >= \beta$	CMP(W) α, β BC $?LFALSE	lower case letters	P.59
	CMP(W) α, β BNC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha >= \beta$ (γ)	MOV(W) γ, α CMP(W) γ, β BC $?LFALSE	lower case letters	
	MOV(W) γ, α CMP(W) γ, β BNC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	

Table B-1. Generated Instructions for Comparison Expressions (3/3)

Comparison expression	Generated instruction	Control statement condition	Page
α <= β	CMP(W) α, β BZ $$+4 BNC $?LFALSE	lower case letters	P.62
	CMP(W) α, β BZ $?LTRUE BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
α <= β (γ)	MOV(W) specified register, α CMP(W) specified register, β BZ $$+4 BNC $?LFALSE	lower case letters	
	MOV(W) specified register, α CMP(W) specified register, β BZ $?LTRUE BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	

γ: specified register

Table B-2. Generated Instructions for Test Bit Expressions

Test bit expression	Generated instruction		Control statement condition	Page
if_bit (bit symbol)	BNC	$?LFALSE	lower case letters (CY)	P.68
elseif_bit (bit symbol)	BNZ	$?LFALSE	lower case letters (Z)	
while_bit (bit symbol)	BF	bit symbol, $?LFALSE	lower case letters	
until_bit (bit symbol)	BC BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters (CY)	
	BZ BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters (Z)	
	BT BR ?LTRUE:	bit symbol, $?LTRUE ?LFALSE	upper case letters	P.69
if_bit (!bit symbol)	BC	$?LFALSE	lower case letters (CY)	P.71
elseif_bit (!bit symbol)	BZ	$?LFALSE	lower case letters (Z)	
while_bit (!bit symbol)	BT	bit symbol, $?LFALSE	lower case letters	
until_bit (!bit symbol)	BNC BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters (CY)	
	BNZ BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters (Z)	
	BF BR ?LTRUE:	bit symbol, $?LTRUE ?LFALSE	upper case letters	P.72

Table B-3. Generated Instructions for Logic Expressions (1/2)

Logic expression	Generated instruction	Control statement condition	Page
α == β &&	CMP(W) α, β BNZ $?LFALSE	lower case letters	P.75, 76
	CMP(W) α, β BZ $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
α != β &&	CMP(W) α, β BZ $?LFALSE	lower case letters	
	CMP(W) α, β BNZ $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
α < β &&	CMP(W) α, β BNC $?LFALSE	lower case letters	
	CMP(W) α, β BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
α > β &&	CMP(W) α, β BZ $?LFALSE BC $?LFALSE	lower case letters	
	CMP(W) α, β BZ $$+4 BNC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
α >= β &&	CMP(W) α, β BC $?LFALSE	lower case letters	
	CMP(W) α, β BNC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
α <= β &&	CMP(W) α, β BZ $$+4 BNC $?LFALSE	lower case letters	
	CMP(W) α, β BZ $?LTRUE BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
CY &&	BNC $?LFALSE	lower case letters	
	BC $?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	

Table B-3. Generated Instructions for Logic Expressions (2/2)

Logic expression	Generated instruction		Control statement condition	Page
Z &&	BNZ	$?LFALSE	lower case letters	P.75, 76
	BZ BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters	
bit symbol &&	BF	bit symbol, $?LFALSE	lower case letters	
	BT BR ?LTRUE:	bit symbol, $?LTRUE ?LFALSE	upper case letters	
!CY &&	BC	$?LFALSE	lower case letters	
	BNC BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters	
!Z &&	BZ	$?LFALSE	lower case letters	
	BNZ BR ?LTRUE:	$?LTRUE ?LFALSE	upper case letters	
!bit symbol &&	BT	bit symbol, $?LFALSE	lower case letters	
	BF BR ?LTRUE:	bit symbol, $?LTRUE ?LFALSE	upper case letters	
α == β ||	CMP(W) BZ	α, β $?LFALSE		P.78
α!= β ||	CMP(W) BNZ	α, β $?LFALSE		
α < β ||	CMP(W) BC	α, β $?LFALSE		
α > β ||	CMP(W) BZ BNC	α, β $?LFALSE $?LFALSE		
α >= β ||	CMP(W) BNC	α, β $?LFALSE		
α <= β ||	CMP(W) BZ BC	α, β $?LFALSE $?LFALSE		
CY ||	BC	$?LFALSE		
Z ||	BZ	$?LFALSE		
bit symbol ||	BT	bit symbol, $?LFALSE		
!CY ||	BNC	$?LFALSE		
!Z ||	BNZ	$?LFALSE		
!bit symbol ||	BF	bit symbol, $?LFALSE		

Table B-4. Expressions (1/4)

Expression	Generated instruction	Page
$\alpha = \beta$	MOV α 1, β	P.85
	MOVW α 1, β	
	BNC ?L1 SET1 α BR ?L2 ?L1: CLR1 α ?L2:	P.84
$\alpha = \beta$ (γ)	MOV γ, β MOV α 1, γ	
	MOVW γ, β MOVW α 1, γ	
	BF β, ?L1 SET1 α BR ?L2 ?L1: CLR1 α ?L2:	
α += β	ADD α, β	P.89
	ADDW α, β	

α += β (γ)	MOV γ, α
	ADD γ, β
	MOV α, γ	

	MOVW γ, α
	ADDW γ, β
	MOVW α, γ	

α += β, CY	ADDC α, β	

α += β, CY (γ)	MOV γ, α
	ADDC γ, β
	MOV α, γ	

Table B-4. Expressions (2/4)

Expression	Generated instruction	Page
α –= β	SUB α, β	P.93
	SUBW α, β	

α –= β (γ)	MOV γ, α
	SUB γ, β
	MOV α, γ	

	MOVW γ, α
	SUBW γ, β
	MOVW α, γ	

α –= β, CY	SUBC α, β	

α –= β, CY (γ)	MOV γ, α
	SUBC γ, β
	MOV α, γ	

α & = β	AND α, β	P.96
	BNC ?L1 BF β, ?L1 SET1 CY BR ?L2 ?L1: CLR1 CY ?L2:	

α &= β (γ)	MOV γ, α
	AND γ, β
	MOV α, γ
		P.97

	BF α, ?L1 BF β, ?L1 SET1 α BR ?L2 ?L1: CLR1 α ?L2:	

Table B-4. Expressions (3/4)

Expression	Generated instruction	Page
α | = β	OR α, β	P.99
	BC ?L1 BF β, ?L2 ?L1: SET1 CY BR ?L3 ?L2: CLR1 CY ?L3:	

α | = β (γ)	MOV γ, α
	OR γ, β
	MOV α, γ
		P.100

	BT α, ?L1 BF β, ?L2 ?L1: SET1 α BR ?L3 ?L2: CLR1 α ?L3:	
α ^ = β	XOR α, β	P.103
	BNC ?L1 BF β, ?L2 ?L1: BC ?L3 BF β, ?L3 ?L2: SET1 CY BR ?L4 ?L3: CLR1 CY ?L4:	

α ^ = β (γ)	MOV γ, α
	XOR γ, β
	MOV α, γ
		P.104

	BF α, ?L1 BF β, ?L2 ?L1: BT α, ?L3 BF β, ?L3 ?L2: SET1 α BR ?L4 ?L3: CLR1 α ?L4:	

Table B-4. Expressions (4/4)

Expression	Generated instruction		Page
α >>= β	ROR : AND	A, 1 A, #0FFH SHR β	P.107
α >>= β (γ)	MOV ROR : AND MOV	A, α A, 1 A, #0FFH SHR β α, A	
α <<= β	ROL : AND	A, 1 A, #LOW (0FFH SHR β)	P.109
α <<= β (γ)	MOV ROL : AND MOV	A, α A, 1 A, #LOW (0FFH SHL β) α, A	
α ++	INC	α	P.111
	INCW	α	
α – –	DEC	α	P.113
	DECW	α	
α <–> β	XCH	α, β	P.115
	XCHW	α, β	
α <–> β (γ)	MOV XCH MOV	γ, α γ, β α, γ	
	MOVW XCHW MOVW	γ, α γ, β α, γ	
α = 1	SET1	α 1	P.118
α = 1 (CY)	SET1 SET1	CY α 1	
α = 0	CLR1	α 1	P.121
α = 0 (CY)	CLR1 CLR1	CY α 1	

APPENDIX C MAXIMUM PERFORMANCE

Table C-1. Maximum Performance of Structured Assembler

Item	Maximum value
Line length (not including LF or CR)	218 characters
Number of symbols registered in #define directive (excluding reserved words)	512 symbols
Nesting levels in control statement	31 levels
Nesting levels in #ifdef directive	8 levels
#defcallt directives	32
Nesting of #include directives	Not supported
Number of redefinitions by #define directive	31 times
Number of operands assigned in a series	33 Note 1
Logical operator operands	17 Note 2
Number of symbols defined by -D option	30

Notes 1. The maximum value is expressed as follows.
S1=S2= ··· S32=S33
Up to 33 symbols, including 32 equal signs (=), can be inserted.

2. The maximum value is expressed as follows.
expression 1$$expression 2&& ··· &&expression 16&&expression 17
Up to 17 expressions and 16 “&&” (or “||”) signs can be inserted.

[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: +55-11-6462-6829

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: 02-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: 044-435-9608

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	❑	❑	❑	❑

CS 00.6