

To our customers,

---

## Old Company Name in Catalogs and Other Documents

---

On April 1<sup>st</sup>, 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 1<sup>st</sup>, 2010  
Renesas Electronics Corporation

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

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

## 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**

# **RA78K4**

## **Structured Assembler Preprocessor**

---

### **ST78K4 V1.10 or Later**

Document No. U11743EJ1V0UMJ1 (1st edition)  
Date Published January 2001 N CP(K)

© NEC Corporation 1996  
Printed in Japan

[MEMO]

**PC DOS™ and IBM PC™ are trademarks of International Business Machines Corp.**

**MS-DOS™ is a trademark of Microsoft Corporation.**

**SunOS™ is a trademark of Sun Microsystems, Inc.**

**HP-UX™ is a trademark of Hewlett-Packard Company.**

**NEWS-OS™ is a trademark of Sony Corporation.**

**PC-9800 is a trademark of NEC Corporation.**

- **The information in this document is current as of September, 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).

# 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 “RA78K4 Assembler Package”, i.e., the assembler package for microcontrollers in the 78K4 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**” (**LANGUAGE, OPERATION**).

### Target readers

This manual was written for readers who understand the functions of compact, general-purpose microcontrollers in the 78K4 Series.

Readers requiring a description of the functions of microcontrollers in the 78K4 Series should refer to the target chip’s User’s Manual.

### Target chips

This assembler can be used for all chips supported by the RA78K4 Assembler Package.

Refer to “Cautions of Usage” of each device file in details.

### Composition

The composition of this manual is as follows.

- CHAPTER 1 presents an overview of the structured assembler.
- CHAPTER 2 to CHAPTER 6 are written as a programming manual. Read these chapters before programming this structured assembler.
- CHAPTER 7 to CHAPTER 11 are written as an operating manual. Read these chapters before running this structured assembler.

- **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.

- **CHAPTER 7 PRODUCT OVERVIEW**

This chapter describes the structured assembler’s files and operating environment.

- **CHAPTER 8 OPERATION METHODS**

This chapter gives detailed descriptions of assembler functions and operation methods.

- **CHAPTER 9 INPUT/OUTPUT FILES**

This chapter describes the format of lists that are input to and output from the structured assembler.

- **CHAPTER 10 ERROR MESSAGES**

This chapter describes error messages that are output by the structured assembler.

- **CHAPTER 11 MAXIMUM PERFORMANCE CHART**

This chapter describes the maximum performance characteristics of the structured assembler.

- **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.

## 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.
<b>Boldface</b>	: 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
$\alpha$	: is entered as a mnemonic operand, such as a register name
$\beta$	: is entered as a mnemonic operand, such as a register name
$\gamma$	: is entered as a mnemonic operand, such as a register name
$\delta$	: is entered as a mnemonic operand, such as a register name



## CONTENTS

<b>CHAPTER 1 GENERAL</b>	<b>1</b>
1.1 Overview of Structured Assembler	1
1.2 Overview of Functions	2
1.3 Before Starting Program Development	4
1.3.1 Maximum performance	4
1.3.2 Caution points	5
1.3.3 Environment variables	6
<b>CHAPTER 2 SOURCE PROGRAM CODING METHODS</b>	<b>7</b>
2.1 Basic Configuration of Source Programs	7
2.2 Source Program Elements	8
2.3 Reserved Words	11
2.4 Label Generation Rules	13
2.5 Size Specification	13
2.6 Data Sizes	14
2.7 Comments	15
2.8 Tool Information	15
<b>CHAPTER 3 CONTROL STATEMENTS</b>	<b>17</b>
3.1 Overview of Control Statements	17
3.2 Control Statement Characters	17
3.3 Nesting	18
3.4 Register Specification	19
3.5 Control Statement Functions	20
3.6 Conditional Expressions	44
3.6.1 Comparison expressions	45
3.6.2 Test bit expressions	67
3.6.3 Logical operations	74
<b>CHAPTER 4 EXPRESSIONS</b>	<b>81</b>
<b>CHAPTER 5 DIRECTIVES</b>	<b>129</b>
5.1 Overview of Directives	129
5.2 Directive Functions	129
<b>CHAPTER 6 CONTROL INSTRUCTIONS</b>	<b>137</b>
6.1 Overview of Control Instructions	137
6.2 Assembler Control Instructions	137
6.3 Control Instruction Functions	140
<b>CHAPTER 7 PRODUCT OVERVIEW</b>	<b>145</b>
7.1 Product Contents	145
7.2 System Configuration	145
7.3 Installation	145

<b>CHAPTER 8 OPERATION METHODS .....</b>	<b>147</b>
8.1 Input/Output File Types .....	147
8.2 Option Functions.....	147
8.3 Launch Method.....	148
8.3.1 Launch of structured assembler .....	148
8.3.2 Execution start and end messages .....	150
8.4 Options.....	151
8.4.1 Types of options .....	151
8.4.2 Option specification method .....	152
8.4.3 Description of options .....	152
<b>CHAPTER 9 INPUT/OUTPUT FILES.....</b>	<b>167</b>
9.1 Input Source Program Files.....	167
9.2 Include Files.....	168
9.3 Secondary Source Program Files .....	169
9.4 Error List Files .....	171
<b>CHAPTER 10 ERROR MESSAGES .....</b>	<b>173</b>
10.1 Overview of Error Messages .....	173
10.2 Abort Errors.....	174
10.3 Fatal Errors .....	177
10.4 Warning Messages.....	180
<b>CHAPTER 11 MAXIMUM PERFORMANCE.....</b>	<b>181</b>
<b>APPENDIX A SYNTAX LISTS .....</b>	<b>183</b>
<b>APPENDIX B LISTS OF GENERATED INSTRUCTIONS.....</b>	<b>187</b>

## List of Figures

Figure No.	Title	Page
1-1.	Structured Assembler Function .....	2
1-2.	Program Development Flowchart .....	3

## List of Tables (1/2)

Table No.	Title	Page
1-1.	Maximum Performance of Structured Assembler .....	4
2-1.	Structured Assembly Language Coding .....	7
2-2.	Alphanumeric Characters .....	8
2-3.	Special Characters .....	9
2-4.	Invalid Characters.....	10
2-5.	Reserved Word Symbols .....	11
2-6.	Data Sizes .....	14
3-1.	Generated Instructions for switch Statement.....	29
3-2.	Comparison Expressions.....	44
3-3.	Test Bit Expressions .....	44
3-4.	Logical Operations.....	44
3-5.	Generated instructions for Comparison Instructions.....	46
3-6.	Generated Instructions (Control Statement in Lower Case Letters) for Logical AND .....	75
3-7.	Generated Instructions (Control Statement in Upper Case Letters) for Logical AND .....	76
3-8.	Generated Instructions for Logical OR .....	78
4-1.	Assignment Statements .....	81
4-2.	Count Statements .....	82
4-3.	Exchange Statements.....	82
4-4.	Bit Manipulation Statements .....	82
4-5.	Generated Instructions for Assignments.....	86
4-6.	Generated Instructions for Increment Assignments.....	90
4-7.	Generated Instructions for Decrement Assignments .....	94
4-8.	Generated Instructions for Multiply Assignments .....	97
4-9.	Generated Instructions for Divide Assignments.....	99
4-10.	Generated Instructions for Logical AND Assignments.....	102
4-11.	Generated Instructions for Logical OR Assignments.....	105
4-12.	Generated Instructions for Logical XOR Assignments .....	108
4-13.	Generated Instructions for RightShiftAssign.....	111
4-14.	Generated Instructions for LeftShiftAssign .....	114
4-15.	Generated Instructions for Increment .....	116
4-16.	Generated Instructions for Decrement.....	118
4-17.	Generated Instructions for Exchange .....	121
4-18.	Generated Instructions for Set Bit.....	124
4-19.	Generated Instructions for Clear Bit .....	127
5-1.	List of Directives .....	129
6-1.	Control Instructions that Can Be Entered Only in Module Headers.....	138
6-2.	Control Instructions that Are Recognized as the Module Body .....	139
6-3.	Control Instruction List.....	140

## List of Tables (2/2)

Table No.	Title	Page
6-4.	Interpretation of Kanji Code .....	143
7-1.	Bundled Files .....	145
8-1.	Structured Assembler's Input/Output Files .....	147
8-2.	Options List .....	151
8-3.	Interpretation of Kanji Code Specifications .....	164
8-4.	Corresponding Control Instructions .....	164
9-1.	Development by Structured Assembler .....	169
10-1.	Abort Error Messages .....	174
10-2.	Fatal Error Messages .....	177
10-3.	Warning Messages .....	180
11-1.	Maximum Performance of Structured Assembler .....	181
A-1.	Control Statements .....	183
A-2.	Conditional Expressions .....	184
A-3.	Expressions .....	184
A-4.	Directives .....	185
B-1.	Generated Instructions for Comparison Expressions .....	187
B-2.	Generated Instructions for Test Bit Expressions .....	190
B-3.	Generated Instructions for Logic Expressions .....	191
B-4.	Expressions .....	193

[MEMO]

## CHAPTER 1 GENERAL

### 1.1 Overview of Structured Assembler

The RA78K4 structured assembler preprocessor is a program in the “RA78K4 Assembler Package” that is used for software development of compact, general-purpose microcontrollers in the 78K/4 Series.

In this manual, the structured assembler preprocessor is abbreviated as the “structured assembler” or the “ST78K4 (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.

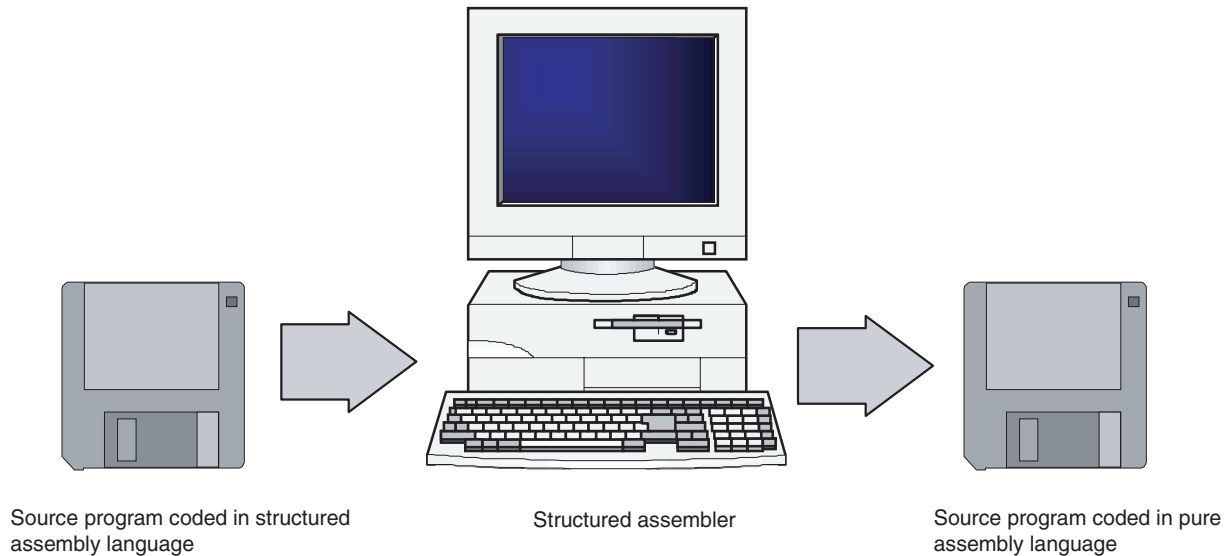
#### <3> Facilitates desktop debugging

- 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.

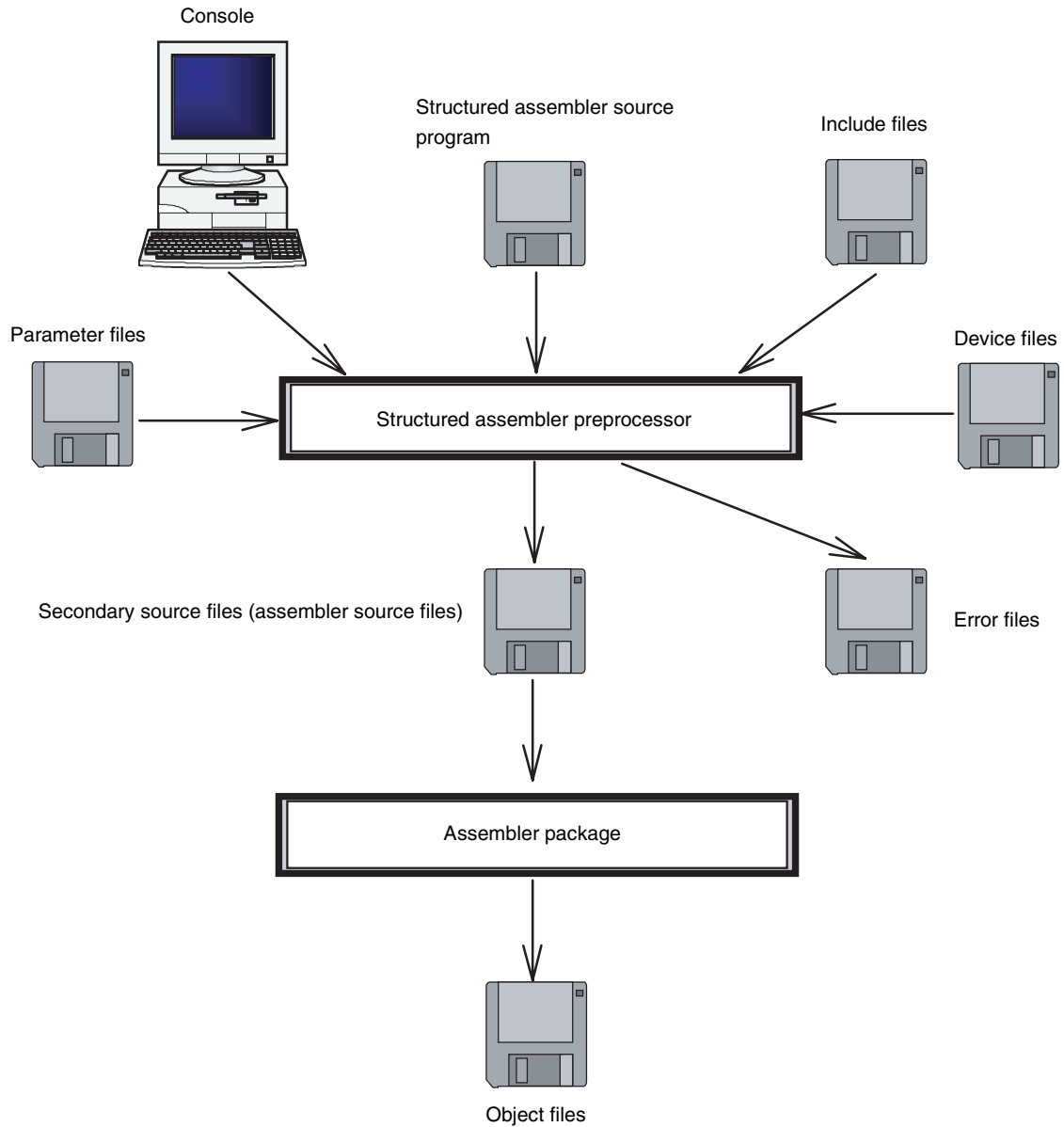
The main functions of the structured assembler are listed below.

- <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 <sup>(Note 1)</sup>
Logical operator operands	17 <sup>(Note 2)</sup>
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.



### 1.3.3 Environment variables

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

#### (1) Coding format

SETΔLANG78K = 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 (\$KANJI CODE)

<3> Specification by LANG78K environment variable

<4> Default specification based on OS

## CHAPTER 2 SOURCE PROGRAM CODING METHODS

### 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 “**RA78K4 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
Structure d assembly	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	Assignment (=), assignment plus operation (+=, etc.), shift (rotate) assignment (>>=, etc.)
		Count statement	Increment (++), decrement (– –)
Exchange statement		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 ST78K4, only the first character in control statements is 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
*	Asterisk	Multiply operation
/	Slash	Divide operation
&	Ampersand	Logical AND operator
	Separator symbol	Logical OR operator
^	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
%	Percent sign	Indirect address specification symbol for 3 bytes or more
!	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	' (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 “**RA78K4 ASSEMBLER PACKAGE USER’S MANUAL LANGUAGE**”



**(5) Expressions**

Expressions are constants, special characters, and symbols that are combined using operators (for details of assembly language expressions, see the “**RA78K4 ASSEMBLER PACKAGE USER'S MANUAL 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, HIGHW, LOWW
	DATAPOS, BITPOS, MASK

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

	Reserved word
Assembler control instructions	PROCESSOR, PC
	DEBAG, NODEBAG, DEBAGA, NODEBAGA
	XREF, XR, NOXREF, NOXR
	TITLE, TI
	SYMLEN, NOSYMLEN
	CAP, NOCAP
	SYMLIST, NOSYMLIST
	FORMFEED, NOFORMFEED
	WIDTH, LENGTH
	TAB
	CHGSFR, CHGSFRA
	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, R8, R9, R10, R11, R12, R13, R14, R15, X, B, C, VPL, VPH, UPL, UPH, D, E, H, L
	PSWL, PSWH, V, U, T, M
	AX
	RP0, RP1, RP2, RP3, RP4, RP5, RP6, RP7, BC, VP, UP, DE, HL
	WHL
	RG4, RG5, RG6, RG7, WP, UUP, TDE
	SP
	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]

( $\Delta$ size\_specification\_character $\Delta$ )

### [Function]

- <1> If the size specification character is “W”, “w”, “P”, or “p”, the data size is changed to bytes.
- <2> 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.

## 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 This structured assembler recognizes bit sfrs and symbols entered using the format “ $\alpha$ . $\beta$ ” as bit symbols. Items that can be entered as “ $\alpha$ ” include byte user symbols, word user symbols, byte-specified user symbols, sfrs, constants, A, X, PSWL, PSWH, [DE], [HL], direct access symbols, and constants. Items that can be entered as “ $\beta$ ” include byte user symbols, word user symbols, and constants.
c	Byte user symbol
d	Byte data Byte data includes byte-specified user symbols and sfrs that overlap saddr2.
e	A
f	Byte registers (R0 to R15, X, B, C, VPL, VPH, UPL, UPH, D, E, H, L)
g	sfr
h	Special registers (PSWL, PSWH, V, U, Y, W)
i	Word user symbol
j	Word data Word data includes word-specified user symbols and sfrps that overlap saddr2.
k	AX
l	Word register (RP0 to RP7, BC, VP, UP, DE, HL)
m	sfrp
n	Triple byte-specified user symbols
o	WHL
p	Triple byte registers (RG4 to RG7, VVP, UUP, TDE)
q	SP
r	Constants
s	Direct access specification symbols These are symbols that are specified using the format “!addr” or “!!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 formats [saddr], [%saddr], [TDE], [WHL], [DE], [HL], [TDE+], [WHL+], [DE+], [HL+], [TDE-], [WHL-], [DE-], [HL-], [VVP], [UUP], [VP], [UP], [TDE+byte], [WHL+byte], [DE+byte], [HL+byte], [SP+byte], [VVP+byte], [UP+byte], [VP+byte], [UP+byte], [TDE+A], [WHL+A], [DE+A], [HL+A], [TDE+B], [WHL+B], [DE+B], [HL+B], [TDE+C], [WHL+C], [DE+C], [HL+C], [VVP+DE], [VVP+HL], [VP+DE], [VP+HL], word[A], word[B], word[DE], and word[HL]. Byte user symbols, word user symbols, constants, and “\$” can be entered as “saddr”, “byte”, or “word”.
u	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. For details of the output information, see the “RA78K4 Assembler Package Operation”.

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 1.10 → 110H

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.

[MEMO]

## CHAPTER 3 CONTROL STATEMENTS

### 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”.

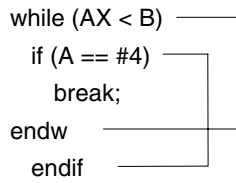
### 3.3 Nesting

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

#### (Example of incorrect coding)

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

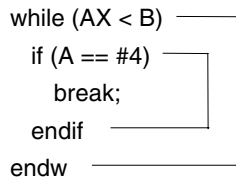
Error occurs due to intersecting.



#### (Example of correct coding)

```
while (AX < 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      R1,#5      ;if (R1!=#5 && R2>=#0 && R3<#80H(R0))
BZ       $?L1
CMP      R2,#0
BC       $?L1
MOV      R0,R3
CMP      R0,#80H
BNC      $?L1
CALL     !XXX          ;CALL !XXX
BR       ?L2
?L1:                      ;else
CALL     !YYY          ;CALL !YYY
?L2:                      ;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      R0,R1      ;if (R1!=#5 && R2>=#0 && R3<#80H ) (R0)
CMP      R0,#5
BZ       $?L3
MOV      R0,R2
CMP      R0,#0
BC       $?L3
MOV      R0,R3
CMP      R0,#80H
BNC      $?L3
CALL     !XXX          ;CALL !XXX
BR       ?L4
?L3:                      ;else
CALL     !YYY          ;CALL !YYY
?L4:                      ;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      R0,R1      ;if (R1!=#5 && R2>=#0(R5) && R3<#80H ) (R0)
CMP      R0,#5
BZ       $?L5
MOV      R5,R2
CMP      R5,#0
BC       $?L5
MOV      R0,R3
CMP      R0,#80H
BNC      $?L5
CALL     !XXX        ;CALL !XXX
BR       ?L6
?L5:
          ;else
CALL     !YYY        ;CALL !YYY
?L6:
          ;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.

---

**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]

## &lt;1&gt; When entered in lower case letters

```

        CMP      A, #0           ; if (A==#0)
        BNZ      $?L1
        MOV1     CY, TFLG.0      ; CY=TFLG.0
        MOVW     AX, TMO        ; AX=TMO
        BR       ?L2
?L1:
        MOVW     BC, #0A00H      ; BC=#0A00H
?L2:
                                     ; endif

```

## &lt;2&gt; When entered in upper case letters

```

        CMP      A, #0           ; IF (A==#0)
        BZ       $?L3
        BR       ?L4
?L3:
        MOV1     CY, TFLG.0      ; CY=TFLG.0
        MOVW     AX, TMO        ; AX=TMO
        BR       ?L5
?L4:
                                     ; ELSE
        MOVW     BC, #0A00H      ; BC=#0A00H
?L5:
                                     ; 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]

## &lt;1&gt; if\_bit~endif

If conditional expression 1 is true, the if\_bit block is executed.

The if\_bit block may occupy several lines.

## &lt;2&gt; 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.

## &lt;3&gt; 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.

## &lt;4&gt; Additional description

These blocks can be mixed in memory with elseif.

## [Description]

## &lt;1&gt; Test bit expressions are entered as conditional expressions 1 and 2.

For details of test bit expressions, see “**3.6 Conditional Expressions**”.

## &lt;2&gt; 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.

## &lt;3&gt; elseif\_bit statements and else statements can be omitted and several elseif\_bit statements can be entered.

---

**Conditional branch****if\_bit**

---

**[Generated instructions]**

## &lt;1&gt; Processing of if\_bit (bit condition)

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

## &lt;2&gt; 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.

## &lt;3&gt; 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.

## &lt;4&gt; 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]

## &lt;1&gt; When entered in lower case letters

```

BT      TRFG.0,$?L1      ;if_bit(!TRFG.0)
MOV     A,#0EH           ;A=#0EH
SET1    PRTYFLG.3        ;PRTYFLG.3=1
BR      ?L2
?L1:
BF      PGF.0,$?L3
MOVW    AX,RDATAP        ;AX=RDATAP
MOVW    BC,#0FFH         ;BC=#0FFH
BR      ?L2
?L3:
MOV     H,#(FG SHR 6)    ;H= #(FG SHR 6)
MOV1    CY,PFG.0         ;CY=PFG.0
CLR1    BUSYFG.2         ;BUSYFG.2=0
?L2:
;endif

```

## &lt;2&gt; When entered in upper case letters

```

BF      TRFG.0,$?L4      ;IF_BIT(!TRFG.0)
BR      ?L5
?L4:
MOV     A,#0EH           ;A=#0EH
SET1    PRTYFLG.3        ;PRTYFLG.3=1
BR      ?L6
?L5:
;ELSEIF_BIT(PGF.0)
BT      PGF.0,$?L7
BR      ?L8
?L7:
MOVW    AX,RDATAP        ;AX=RDATAP
MOVW    BC,#0FFH         ;BC=#0FFH
BR      ?L6
?L8:
;ELSE
MOV     H,#(FG SHR 6)    ;H= #(FG SHR 6)
MOV1    CY,PFG.0         ;CY=PFG.0
CLR1    BUSYFG.2         ;BUSYFG.2=0
?L6:
;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]**

## &lt;1&gt; 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.**

## &lt;2&gt; 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

CMP(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

## &lt;3&gt; Processing of default statement

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

## &lt;4&gt; Processing of ends statement

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

Conditional branch

switch

Table 3-1. Generated Instructions for switch Statement

CASE symbol		Without register specification	With register specification												
			CY	d	e	f	g	h	j	k	l	m	o	p	q
a	CY														
b	Bit symbol														
c	Byte user symbol	*3		*4	*1	*1	*1		*5	*2	*2	*2			
d	Byte data	*3		*4	*1	*1	*1								
e	A	*3		*4	*1	*1	*1								
f	Byte register	*3		*4	*1	*1	*1								
g	sfr	*3		*4	*1	*1	*1								
h	Special register	*1			*1										
i	Word user symbol	*3		*4	*1	*1	*1		*5	*2	*2	*2			
j	Word data	*3							*5	*2	*2	*2			
k	AX	*3							*5	*2	*2	*2			
l	Word register	*2							*5	*2	*2	*2			
m	sfrp	*3							*5	*2	*2	*2			
n	Triple byte specification														
o	WHL														
p	Triple byte register														
q	SP														
r	Constant	*3		*4	*1	*1	*1		*5	*2	*2	*2			
s	Direct access symbol	*1			*1	*1				*2					
t	Indirect access symbol	*1			*1					*2					
u	Immediate symbol	*1		*4	*1	*1	*1		*5	*2	*2	*2			

\*1 : Generates MOV instruction

\*2 : Generates MOVW instruction

\*3 : Does not generate transfer instruction

\*4 : The register specification's byte data is an sfr that overlaps the saddr2 area.

\*5 : The register specification's word data is an sfrp that overlaps the saddr2 area.

Empty columns indicate errors.



**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]

## &lt;1&gt; 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

```

## &lt;2&gt; 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]

## &lt;1&gt; When entered in lower case letters

```

?L1:                                ;while (AX<#0FFFH)
        CMPW    AX,#0FFFH
        BNC     $?L2
        CMP     B,#0                ; if (B==#0)
        BNZ     $?L3
        BR      ?L2                ; break
?L3:                                ; endif
        INCW    HL                ; HL++
        BR      ?L1
?L2:                                ;endw

```

## &lt;2&gt; When entered in upper case letters

```

?L4:                                ;WHILE (AX<#0FFFH)
        CMPW    AX,#0FFFH
        BC      $?L5
        BR      ?L6
?L5:
        CMP     B,#0                ; if (B==#0)
        BNZ     $?L7
        BR      ?L6                ; break
?L7:                                ; endif
        INCW    HL                ; HL++
        BR      ?L4
?L6:                                ;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]

## &lt;1&gt; 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

```

## &lt;2&gt; 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
        CLR1         PFG.0          ; PFG.0=0
?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]

## &lt;1&gt; 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

```

## &lt;2&gt; 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
                   ; B=#8H
      MOV          B,#8H
      CALL         !XXX      ; CALL !XXX
      BF           TRFG.0,$?L1 ;until_bit(TRFG.0)

```

**<2> When entered in upper case letters**

```

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

```

Conditional loop

break

**(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     ?Lxxx

**[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 (X==#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

```



**Conditional loop****goto****(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:	MOV B, #0 MOV PORT4, A CMP A, #0FH BNZ \$?L2 BR ERROR	;while(forever) ; B=#0 ; PORT4=A ; if (A==#0FH) ; goto ERROR
?L2:	INCW HL BR ?L1	; endif ; HL++ ;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	$\alpha == \beta$	True when $\alpha = \beta$ , false when $\alpha \neq \beta$
(2)	NotEqual	$\alpha != \beta$	True when $\alpha \neq \beta$ , false when $\alpha = \beta$
(3)	LessThan	$\alpha < \beta$	True when $\alpha < \beta$ , false when $\alpha \geq \beta$
(4)	GreaterThan	$\alpha > \beta$	True when $\alpha > \beta$ , false when $\alpha \leq \beta$
(5)	GreaterEqual	$\alpha \geq \beta$	True when $\alpha \geq \beta$ , false when $\alpha < \beta$
(6)	LessEqual	$\alpha \leq \beta$	True when $\alpha \leq \beta$ , 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 ( $\gamma$ ) is specified at the end of a comparison, a comparison can be made between  $\alpha$  and  $\beta$  values that cannot be compared directly.

$\gamma$  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

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY																		*3			
	b	Bit symbol																		*3			
	c	Byte user symbol			*1	*1	*1	*1				*2	*2	*2						*1			*1
	d	Byte data			*1	*1	*1	*1			*1									*1			*1
	e	A			*1	*1	*1	*1	*1		*1									*1	*1	*1	*1
	f	Byte register			*1	*1	*1	*1	*1		*1									*1			*1
	g	sfr					*1	*1												*1			*1
	h	Special register																					
	i	Word user symbol				*1	*1	*1				*2	*2	*2	*2					*2			*2
	j	Word data			*2							*2	*2	*2	*2					*2			*2
	k	AX			*2							*2	*2	*2	*2	*2				*2			*2
	l	Word register			*2							*2	*2	*2	*2	*2				*2			*2
	m	sfrp											*2	*2									*2
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
	q	SP																					
	r	Constant			*1	*1	*1	*1				*2	*2	*2	*2					*1			*1
	s	Direct access symbol					*1																
	t	Indirect access symbol					*1																
	u	Immediate symbol																					

\*1 : Generates CMP instruction

\*2 : Generates CMPW instruction

\*3 : If "1" is specified as  $\beta$  and if the comparison expression includes ==, then the generated instruction is the same as for the bit symbol.

If "1" is specified as  $\beta$  and if the comparison expression includes !=, then the generated instruction is the same as for the !bit symbol.

If "0" is specified as  $\beta$  and if the comparison expression includes ==, then the generated instruction is the same as for the !bit symbol.

If "0" is specified as  $\beta$  and if the comparison expression includes !=, then the generated instruction is the same as for the bit symbol.

For details, see "3.6.2 (1) Bit symbol" and "3.6.2 (2) !bit symbol".

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  $\alpha$  and  $\beta$  are equal, false when they are not equal.

**<2> When there is a register specification**

The contents of  $\alpha$  are transferred to the specified register. True is the result when the contents of the specified register are equal to the contents of  $\beta$  and false is the result when they are not equal.

**[Description]****<1> When there is no register specification**

For  $\alpha$  and  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

**<2> When there is a register specification**

For  $\alpha$ , be sure to specify contents that can be entered in MOV or MOVW.

For  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

**<3> If  $\alpha$  is a bit symbol**

If  $\beta = 1$ , the processing is the same as for a test bit expression's bit symbol.

If  $\beta = 0$ , the processing is the same as for a test bit expression's !bit symbol.

For details of test bit expressions, see "**3.6.2 Test bit expressions**".

**[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 (==)

## &lt;3&gt; If the control statement is entered in upper case letters and there is no register specification

```

CMP(W)     $\alpha$ ,  $\beta$ 
BZ        $?LTRUE
BR        ?LFALSE
?LTRUE:

```

## &lt;4&gt; If the control statement is entered in upper case letters and there is a register specification

```

MOV(W)    Specified register,  $\alpha$ 
CMP(W)    Specified register,  $\beta$ 
BZ        $?LTRUE
BR        ?LFALSE
?LTRUE:

```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 3-5. Generated instructions for Comparison Instructions**.  $\alpha$  indicates the specified register. For further description of generated instructions for MOV, see “**CHAPTER 4 (1) Assign**”.

## [Use examples]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

```

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

```

## &lt;2&gt; 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 (==)

## &lt;3&gt; If the control statement is entered in upper case letters and there is no register specification

```

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

```

## &lt;4&gt; 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

```

## Comparison expressions

## NotEqual (!=)

## (2) NotEqual (!=)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

True when the contents of  $\alpha$  and  $\beta$  are not equal, false when they are equal.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  are transferred to the specified register. True is the result when the contents of the specified register are not equal to the contents of  $\beta$  and false is the result when they are equal.

## [Description]

## &lt;1&gt; When there is no register specification

For  $\alpha$  and  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## &lt;2&gt; When there is a register specification

For  $\alpha$ , specify contents that can be entered in MOV or MOVW.

For  $\beta$ , specify contents that can be entered in CMP or CMPW.

<3> If  $\alpha$  is a bit symbol

If  $\beta = 1$ , the processing is the same as for a test bit expression's bit symbol.

If  $\beta = 0$ , the processing is the same as for a test bit expression's !bit symbol.

For details of test bit expressions, see "3.6.2 Test bit expressions".

## [Generated instructions]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

CMP(W)             $\alpha, \beta$

BZ                \$?LFALSE

## &lt;2&gt; If the control statement is entered in lower case letters and there is a register specification

MOV(W)            Specified register,  $\alpha$

CMP(W)            Specified register,  $\beta$

BZ                \$?LFALSE



## Comparison expressions

NotEqual (!=)

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

```

CMP(W)     $\alpha, \beta$ 
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,  $\alpha$ 
CMP(W)     Specified register,  $\beta$ 
BNZ        $?LTRUE
BR         ?LFALSE
?LTRUE:

```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 3-5. Generated instructions for Comparison Instructions**.  $\alpha$  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, HL           ; if (AX!=HL)
BZ        $?L1
CALL      !XXX             ; CALL !XXX
BR        ?L2
?L1:
CALL      !YYY             ; else
                                ; 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

```

## Comparison expressions

## NotEqual (!=)

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

```

      CMPW    AX,HL           ; IF (AX!=HL)
      BNZ     $?L4
      BR      ?L5

?L4:
      CALL    !XXX           ; CALL !XXX
      BR      ?L6

?L5:
      CALL    !YYY           ; CALL !YYY
      BR      ?L6

?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

```

Comparison expressions

LessThan (&lt;)

**(3) LessThan (<)****[Coding format]**

$$[\Delta] \text{ [size specification] } [\Delta] \alpha [\Delta] < [\Delta] \text{ [size specification] } [\Delta] \beta [\Delta] \text{ [(register specification)]}$$
**[Function]****<1> When there is no register specification**

True when the contents of  $\alpha$  are less than the contents of  $\beta$ , false when otherwise (i.e., equal to or greater than).

**<2> When there is a register specification**

The contents of  $\alpha$  are transferred to the specified register. True is the result when the contents of the specified register are less than the contents of  $\beta$  and false is the result when they are otherwise.

**[Description]****<1> When there is no register specification**

For  $\alpha$  and  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

**<2> When there is a register specification**

For  $\alpha$ , be sure to specify contents that can be entered in MOV or MOVW.

For  $\beta$ , 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)	$\alpha, \beta$
BNC	\$?LFALSE

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

MOV(W)	Specified register, $\alpha$
CMP(W)	Specified register, $\beta$
BNC	\$?LFALSE

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

CMP(W)	$\alpha, \beta$
BC	\$?LTRUE
BR	?LFALSE

?LTRUE:

## Comparison expressions

LessThan (&lt;)

## &lt;4&gt; If the control statement is entered in upper case letters and there is a register specification

```

MOV(W)      Specified register,  $\alpha$ 
CMP(W)      Specified register,  $\beta$ 
BC           $\$?LTRUE$ 
BR           $?LFALSE$ 
?LTRUE:

```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 3-5. Generated instructions for Comparison Instructions**.  $\alpha$  indicates the specified register. For further description of generated instructions for MOV, see “**CHAPTER 4 (1) Assign**”.

## [Use examples]

## &lt;1&gt; 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:
CALL     !YYY             ; else
                           ; CALL  !YYY
?L2:
                           ;endif

```

## &lt;2&gt; If the control statement is entered in lower case letters and there is a register specification

```

MOVW     AX,ABCP          ; if (ABCP<HL (AX) )
CMPW     AX,HL
BNC       $\$?L3$ 
CALL     !PPP             ; CALL  !PPP
?L3:
                           ;endif

```

## &lt;3&gt; 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<HL (AX) )
CMPW    AX,HL
BC       $?L7
BR       ?L8
?L7:
CALL    !PPP                   ; CALL    !PPP
?L8:                                ;ENDIF

```

## Comparison expressions

## GreaterThan (&gt;)

## (4) GreaterThan (&gt;)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

True when the contents of  $\alpha$  are greater than the contents of  $\beta$ , false when otherwise (i.e. equal to or less than).

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  are transferred to the specified register. True is the result when the contents of the specified register are greater than the contents of  $\beta$  and false is the result when they are otherwise.

## [Description]

## &lt;1&gt; When there is no register specification

For  $\alpha$  and  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## &lt;2&gt; When there is a register specification

For  $\alpha$ , be sure to specify contents that can be entered in MOV or MOVW.

For  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## [Generated instructions]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

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

## &lt;2&gt; 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
BNH          $?LFALSE
```

## Comparison expressions

GreaterThan (&gt;)

## &lt;3&gt; If the control statement is entered in upper case letters and there is no register specification

```

CMP(W)     $\alpha, \beta$ 
BH        $?LTRUE
BR        ?LFALSE
?LTRUE:

```

## &lt;4&gt; If the control statement is entered in upper case letters and there is a register specification

```

MOV(W)    Specified register,  $\alpha$ 
CMP(W)    Specified register,  $\beta$ 
BH        $?LTRUE
BR        ?LFALSE
?LTRUE:

```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 3-5. Generated instructions for Comparison Instructions**.  $\alpha$  indicates the specified register. For further description of generated instructions for MOV, see “**CHAPTER 4 (1) Assign**”.

## [Use examples]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

```

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

```

## &lt;2&gt; If the control statement is entered in lower case letters and there is a register specification

```

MOVW     AX, ABCP          ; if (ABCP>HL(AX))
CMPW     AX, HL
BNH      $?L3
CALL     !PPP              ; CALL !PPP
?L3:
                           ; endif

```

## Comparison expressions

## GreaterThan (&gt;)

## &lt;3&gt; If the control statement is entered in upper case letters and there is no register specification

```

        CMP    A, [HL]          ; IF (A>[HL])
        BH     $?L4
        BR     ?L5

?L4:
        CALL   !XXX             ; CALL !XXX
        BR     ?L6

?L5:
                                   ; ELSE
        CALL   !YYY             ; CALL !YYY
?L6:
                                   ; ENDIF

```

## &lt;4&gt; If the control statement is entered in upper case letters and there is a register specification

```

        MOVW    AX, ABCP         ; IF (ABCP>HL(AX))
        CMPW    AX, HL
        BH     $?L7
        BR     ?L8

?L7:
        CALL    !PPP             ; CALL !PPP
?L8:
                                   ; ENDIF

```



## Comparison expressions

## GreaterEqual (&gt;=)

## (5) GreaterEqual (&gt;=)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

True when the contents of  $\alpha$  are greater than or equal to the contents of  $\beta$ , false when they are less than the contents of  $\beta$ .

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  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  $\beta$  and false is the result when they are less than the contents of  $\beta$ .

## [Description]

## &lt;1&gt; When there is no register specification

For  $\alpha$  and  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## &lt;2&gt; When there is a register specification

For  $\alpha$ , be sure to specify contents that can be entered in MOV or MOVW.

For  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## [Generated instructions]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

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

## &lt;2&gt; 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
```

## &lt;3&gt; 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,  $\alpha$ 
CMP(W)    Specified register,  $\beta$ 
BNC       $?LTRUE
BR        ?LFALSE
?LTRUE:
```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 3-5. Generated Instructions for Comparison Instructions**.  $\alpha$  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:
CALL     !YYY             ; else
                          ; 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] >=HL (AX) )
CMPW     AX, HL
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] >=HL (AX) )
CMPW    AX, HL
BNC      $?L7
BR       ?L8
?L7:
CALL    !PPP                    ; CALL    !PPP
?L8:                                     ;ENDIF
```

## Comparison expressions

## LessEqual (&lt;=)

## (6) LessEqual (&lt;=)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

True when the contents of  $\alpha$  are less than or equal to the contents of  $\beta$ , false when they are greater than the contents of  $\beta$ .

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  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  $\beta$  and false is the result when they are greater than the contents of  $\beta$ .

## [Description]

## &lt;1&gt; When there is no register specification

For  $\alpha$  and  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## &lt;2&gt; When there is a register specification

For  $\alpha$ , be sure to specify contents that can be entered in MOV or MOVW.

For  $\beta$ , be sure to specify contents that can be entered in CMP or CMPW.

## [Generated instructions]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

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

## &lt;2&gt; 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
BH          $?LFALSE
```

## &lt;3&gt; If the control statement is entered in upper case letters and there is no register specification

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

## Comparison expressions

LessEqual (&lt;=)

## &lt;4&gt; If the control statement is entered in upper case letters and there is a register specification

```

MOV(W)      Specified register,  $\alpha$ 
CMP(W)      Specified register,  $\beta$ 
BNH         $?LTRUE
BR          ?LFALSE
?LTRUE:

```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 3-5. Generated Instructions for Comparison Instructions**.  $\alpha$  indicates the specified register. For further description of generated instructions for MOV, see “**CHAPTER 4 (1) Assign**”.

## [Use examples]

## &lt;1&gt; If the control statement is entered in lower case letters and there is no register specification

```

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

```

## &lt;2&gt; If the control statement is entered in lower case letters and there is a register specification

```

      MOVW    AX, [DE]          ; if ( [DE] <=HL (AX) )
      CMPW    AX, HL
      BH      $?L3
      CALL    !PPP              ; CALL    !PPP
?L3:
                                   ;endif

```

## &lt;3&gt; If the control statement is entered in upper case letters and there is no register specification

```

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

```

## Comparison expressions

## LessEqual (&lt;=)

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

```

MOVW    AX, [DE]          ; IF ( [DE] <=HL (AX) )
CMPW    AX, HL
BNH     $?L7
BR      ?L8
?L7:
CALL    !PPP              ; CALL !PPP
?L8:
;ENDIF

```

Comparison expressions

FOREVER (forever)

### (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:
INC      i
BR       ?L1                ; endif
?L3:
;next

```

##### <2> while statement

```

?L4:
BF       forever, $?L5       ;while (forever)
MOV      A, i                ; A=i
CALL     !XX                 ; CALL !XX
CMPW     AX, #0FFH           ; if (AX==#0FFH)
BNZ      $?L6
BR       ?L5                ; break
?L6:
INC      i
BR       ?L4                ; endif
?L5:
;endw <2> while statement

```

---

**Comparison expressions**

---

**FOREVER (forever)**

---

**<3> repeat statement**

```
?L7:      MOV      A,i          ;repeat
          CALL     !XXX        ; A=i
          CMPW     AX,#0FFH     ; CALL !XXX
          BNZ      $?L8         ; if (AX==#0FFH)
          BR       ?L9          ; break
?L8:      INC      i           ; endif
          BR       ?L7          ; i++
?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          for
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:
          CALL     !YYY          ; CALL  !YYY
?L2:
          BNC      $?L3          ;if_bit(Z)
          CALL     !XXX          ; CALL  !XXX
          BR       ?L4
?L3:
          CALL     !YYY          ; CALL  !YYY
?L4:
          BNC      TRFG.0,$?L5   ;if_bit(TRFG.0)
          CALL     !XXX          ; CALL  !XXX
          BR       ?L6
?L5:
          CALL     !YYY          ; CALL  !YYY
?L6:

```

## Test bit expressions

## Positive logic (bit)

## &lt;2&gt; 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:
      CALL    !YYY      ; CALL !YYY
      ;ELSE
?L9:
      ;ENDIF

      BZ      $?L10     ; IF_BIT(Z)
      BR      ?L11

?L10:
      CALL    !XXX      ; CALL !XXX
      BR      ?L12

?L11:
      CALL    !YYY      ; CALL !YYY
      ;ELSE
?L12:
      ;ENDIF

      BT      TRFG.0,$?L13 ; IF_BIT(TRFG.0)
      BR      ?L14

?L13:
      CALL    !XXX      ; CALL !XXX
      BR      ?L15

?L14:
      CALL    !YYY      ; CALL !YYY
      ;ELSE
?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          for
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:
CALL        !YYY          ; CALL !YYY
?L2:
endif

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

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

```

## Test bit expressions

## Negative logic (bit)

## &lt;2&gt; 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:      CALL  !YYY          ; ELSE
      CALL  !YYY          ; CALL  !YYY
?L9:      ;ENDIF

      BNZ      $?L10         ; IF_BIT(!Z)
      BR       ?L11

?L10:     CALL  !XXX          ; CALL  !XXX
      BR       ?L12

?L11:     CALL  !YYY          ; 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:     CALL  !YYY          ; 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]

```
X<#0FFH && X>=#0 || A==#10
```

The logical operations are described below.



**Logical operations****Logical AND (&&)****(1) Logical AND (&&)****[Coding format]**

```
Conditional expression 1 [ $\Delta$ ] && [ $\Delta$ ] 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
$\alpha == \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNZ \$?LFALSE
$\alpha != \beta \ \&\&$	CMP(W) $\alpha, \beta$ BZ \$?LFALSE
$\alpha < \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNC \$?LFALSE
$\alpha > \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNH \$?LFALSE
$\alpha >= \beta \ \&\&$	CMP(W) $\alpha, \beta$ BC \$?LFALSE
$\alpha <= \beta \ \&\&$	CMP(W) $\alpha, \beta$ BH \$?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

## Logical operations

## Logical AND (&amp;&amp;)

<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
$\alpha == \beta \ \&\&$	CMP(W) $\alpha, \beta$ BZ \$?LTRUE BR ?LFALSE ?LTRUE:
$\alpha != \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNZ \$?LTRUE BR ?LFALSE ?LTRUE:
$\alpha < \beta \ \&\&$	CMP(W) $\alpha, \beta$ BC \$?LTRUE BR ?LFALSE ?LTRUE:
$\alpha > \beta \ \&\&$	CMP(W) $\alpha, \beta$ BH \$?LTRUE BR ?LFALSE ?LTRUE:
$\alpha >= \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNC \$?LTRUE BR ?LFALSE ?LTRUE:
$\alpha <= \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNH \$?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 (&amp;&amp;)

**[Use examples]****<1> When the control statement is entered in lower case letters**

```

        CMP     A, #0                ; if (A==#0 && B>=#0 && B<#80H)
        BNZ     $?L1
        CMP     B, #0
        BC      $?L1
        CMP     A, #80H
        BNC     $?L1
        CALL    !XXX                ; CALL    !XXX
        BR      ?L2

?L1:
        CALL    !YYY                ; else
                                   ; CALL    !YYY
?L2:
                                   ; endif

```

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

```

        CMP     A, #0                ; IF (A==#0 && B>=#0 && B<#80H)
        BZ      $?L3
        BR      ?L6

?L3:
        CMP     B, #0
        BNC     $?L4
        BR      ?L6

?L4:
        CMP     B, #80H
        BC      $?L5
        BR      ?L6

?L5:
        CALL    !XXX                ; CALL    !XXX
        BR      ?L7

?L6:
                                   ; ELSE
        CALL    !YYY                ; CALL    !YYY
?L7:
                                   ; ENDIF

```

## Logical operations

## Logical OR (||)

## (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
$\alpha == \beta$	CMP(W) $\alpha, \beta$ BZ \$?LFALSE
$\alpha != \beta$	CMP(W) $\alpha, \beta$ BNZ \$?LFALSE
$\alpha < \beta$	CMP(W) $\alpha, \beta$ BC \$?LFALSE
$\alpha > \beta$	CMP(W) $\alpha, \beta$ BH \$?LFALSE
$\alpha >= \beta$	CMP(W) $\alpha, \beta$ BNC \$?LFALSE
$\alpha <= \beta$	CMP(W) $\alpha, \beta$ BNH \$?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 (II)

[Use examples]

```

CMP    A,#0                ;if (A==#0 || B>=#0 || B<#80H)
BZ     $?L1
CMP    B,#0
BNC    $?L1
CMP    B,#80H
BNC    $?L2
?L1:
CALL   !XXX                ; CALL !XXX
BR     ?L3
?L2:
CALL   !YYY                ;else
                                ; 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	$\alpha 1 = \dots = \alpha n = \beta$	$\alpha 1 \leftarrow \beta, \dots, \alpha n \leftarrow \beta$
	Sequential assign (with register specification)	$\alpha 1 = \dots = \alpha n = \beta (\gamma)$	$\gamma \leftarrow \beta, \alpha 1 \leftarrow \gamma, \dots, \alpha n \leftarrow \gamma$
(2)	Increment assignment	$\alpha += \beta$	$\alpha \leftarrow \alpha + \beta$
	Increment assignment (with register specification)	$\alpha += \beta$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma + \beta, \alpha \leftarrow \gamma$
	Increment assignment (with register specification)	$\alpha += \beta, CY$	$\alpha \leftarrow \alpha + \beta, CY$
	Increment assignment (with register specification)	$\alpha += \beta, CY$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma + \beta, CY, \alpha \leftarrow \gamma$
(3)	Decrement assignment	$\alpha -= \beta$	$\alpha \leftarrow \alpha - \beta$
	Decrement assignment (with register specification)	$\alpha -= \beta$ , (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma - \beta, \alpha \leftarrow \gamma$
	Decrement assignment (with register specification)	$\alpha -= \beta, CY$	$\alpha \leftarrow \alpha - \beta, CY$
	Decrement assignment (with register specification)	$\alpha -= \beta, CY$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma - \beta, CY, \alpha \leftarrow \gamma$
(4)	Multiply assignment	$\alpha *= \beta$	$\alpha \leftarrow \alpha \times \beta$
	Multiply assignment (with register specification)	$\alpha *= \beta$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \times \beta, \alpha \leftarrow \gamma$
(5)	Divide assignment	$\alpha /= \beta$	$\alpha \leftarrow \alpha \div \beta$
(6)	Logical AND assignment	$\alpha \&= \beta$	$\alpha \leftarrow \alpha \cap \beta$
	Logical AND assignment (with register specification)	$\alpha \&= \beta$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \cap \beta, \alpha \leftarrow \gamma$
(7)	Logical OR assignment	$\alpha  = \beta$	$\alpha \leftarrow \alpha \cup \beta$
	Logical OR assignment (with register specification)	$\alpha  = \beta$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \cup \beta, \alpha \leftarrow \gamma$
(8)	Logical XOR assignment	$\alpha \wedge= \beta$	$\alpha \leftarrow \alpha \wedge \beta$
	Logical XOR assignment (with register specification)	$\alpha \wedge= \beta$ (register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \wedge \beta, \alpha \leftarrow \gamma$
(9)	Right shift (rotate) assignment	$\alpha >>= \beta$	( $\alpha$ shifted to right of $\beta$ bit)
	Right shift assignment (with register specification)	$\alpha >>= \beta$ (register)	$\gamma \leftarrow \alpha, (\gamma$ shifted to right of $\beta$ bit), $\alpha \leftarrow \gamma$
(10)	Left shift assignment	$\alpha <<= \beta$	( $\alpha$ shifted to left of $\beta$ bit)
	Left shift assignment (with register specification)	$\alpha <<= \beta$ (register)	$\gamma \leftarrow \alpha, (\gamma$ shifted to left of $\beta$ bit), $\alpha \leftarrow \gamma$

Table 4-2. Count Statements

Count statement		Coding format	Function
(11)	Increment	$\alpha++$	$\alpha \leftarrow \alpha + 1$
(12)	Decremen	$\alpha--$	$\alpha \leftarrow \alpha - 1$

Table 4-3. Exchange Statements

Exchange statement		Coding format	Function
(13)	Exchange	$\alpha <-> \beta$	$\alpha \leftarrow \alpha <-> \beta$
	Exchange (with register specification)	$\alpha <-> \beta (\gamma)$	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma <-> \beta, \alpha \leftarrow \gamma$

Table 4-4. Bit Manipulation Statements

Bit manipulation statement		Coding format	Function
(14)	Set bit	$\alpha = 1$	$\alpha \leftarrow 1$
	Set bit (with register specification)	$\alpha = 1 \text{ (CY)}$	$\text{CY} \leftarrow 1, \alpha \leftarrow 1$
	Sequential set bit	$\alpha 1 = \dots = \alpha n = 1$	$\alpha n \leftarrow 1, \dots, \alpha 1 \leftarrow 1$
	Sequential set bit (with register specification)	$\alpha 1 = \dots = \alpha n = 1 \text{ (CY)}$	$\text{CY} \leftarrow 1, \alpha n \leftarrow 1, \dots, \alpha 1 \leftarrow 1$
(15)	Clear bit	$\alpha = 0$	$\alpha \leftarrow 0$
	Clear bit (with register specification)	$\alpha = 0 \text{ (CY)}$	$\text{CY} \leftarrow 0, \alpha \leftarrow 0$
	Sequential clear bit	$\alpha 1 = \dots = \alpha n = 0$	$\alpha n \leftarrow 0, \dots, \alpha 1 \leftarrow 0$
	Sequential clear bit (with register specification)	$\alpha 1 = \dots = \alpha n = 0 \text{ (CY)}$	$\text{CY} \leftarrow 0, \alpha n \leftarrow 0, \dots, \alpha 1 \leftarrow 0$

The functions of these expressions are described below.



**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**

$\beta$  values on the right side are sequentially assigned to the left side.

**<2> When there is a register specification**

$\beta$  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]**

$\alpha$  and  $\beta$  are values that can be entered via the MOV1, MOV, MOVW, or MOVG 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 register specification**

```
MOV      α 1, β
```

MOV1, MOVW, MOVG may be generated instead, depending on the operand.

**<2> When there is no register specification and a sequential assignment has been entered**

```
MOV      α n, β
```

```
MOV      α n-1, β
```

```
:
```

```
MOV      α 2, β
```

```
MOV      α 1, β
```

MOV1, MOVW, MOVG may be generated instead, depending on the operand.

Assignment statements	Assign (=)
<b>&lt;3&gt; When there is a register specification</b>	
MOV	specified register, $\alpha$
MOVW	$\alpha$ 1, $\beta$
MOV1, MOVW, MOVG may be generated instead, depending on the operand.	
<b>&lt;4&gt; When there is a register specification and a sequential assignment has been entered</b>	
MOV	specified register, $\beta$
MOV	$\alpha$ n, specified register
MOV	$\alpha$ n-1, specified register
:	
MOV	$\alpha$ 2, specified register
MOV	$\alpha$ 1, specified register
MOV1, MOVW, MOVG may be generated instead, depending on the operand.	

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-5. Generated Instructions for Assignments**. Depending on the entered statement,  $\alpha$  n or  $\beta$  indicates the specified register.

#### [Use examples]

##### <1> When there is no register specification

```

MOV1    CY, P1.1      ;CY=P1.1
MOV     A, #4H        ;A=#4H
MOVW    AX, SYMP      ;AX=SYMP
MOVG    VVP, UUP      ;VVP=UUP
MOVG    SP, #4FFFFH   ;SP=#4FFFFH
MOV1    [HL].1, CY     ;PORT.0=X.2=[HL].1=CY
MOV1    X.2, CY
MOV1    PORT.0, CY
MOV     DAT3, X        ;DAT1=DAT2=DAT3=X
MOV     DAT2, X
MOV     DAT1, X
MOVW    DATA3P, BC    ;DATA1P=DATA2P=DATA3P=BC
MOVW    DATA2P, BC
MOVW    DATA1P, BC
MOV     TDATA3G, #0FFFFH ;TDATA1G=TDATA2G=TDATA3G=#0FFFFH
MOV     TDATA2G, #0FFFFH
MOV     TDATA1G, #0FFFFH

```

## Assignment statements

## Assign (=)

## &lt;2&gt; When there is a register specification

```

MOV1      CY, P1.1      ; A.0=P1.1 (CY)
MOV1      A.0, CY
MOV       A, #4H        ; [DE] = #4H (A)
MOV       [DE], A
MOVW     AX, SYMP       ; BC=SYMP (AX)
MOVW     BC, AX
MOV1     CY, [HL].1     ; PORT.0=X.2=[HL].1 (CY)
MOV1     X.2, CY
MOV1     PORT.0, CY
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
MOVG     WHL, TDATA3G   ; TDATA1G=TDATA2G=TDATA3G (WHL)
MOVG     TDATA2G, WHL
MOVG     TDATA1G, WHL

```

Assignment statements

Assign (=)

Table 4-5. Generated Instructions for Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY		*4	*4						*4									*5			
	b	Bit symbol		*4																*5			
	c	Byte user symbol		*4		*1	*1	*1	*1				*2	*2	*2			*3	*3	*6			*1
	d	Byte data				*1	*1	*1	*1			*1								*1			*1
	e	A				*1	*1	*1	*1	*1	*1	*1								*1	*1	*1	*1
	f	Byte register				*1	*1	*1	*1	*1		*1								*1	*1		*1
	g	sfr							*1	*1													*1
	h	Special register							*1														*1
	i	Word user symbol																					
	j	Word data																					
	k	AX																					
	l	Word register																					
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
q	SP																						
r	Constant																						
s	Direct access symbol																						
t	Indirect access symbol																						
u	Immediate symbol																						

\*1 : Generates MOVCMP

\*2 : Generates MOVW

\*3 : Generates MOVG

\*4 : Generates MOV1

\*5 : Generates SET1 if  $\beta$  is specified as 1. Generates CLR1 if  $\beta$  is specified as 0.\*6 : Generates SET1 if  $\beta$  is specified as 1. Generates CLR1 if  $\beta$  is specified as 0. Generates MOV if  $\beta$  is specified as other than 0 or 1.\*7 : Generates SET1 if  $\beta$  is specified as 1. Generates CLR1 if  $\beta$  is specified as 0. Generates MOVW if  $\beta$  is specified as other than 0 or 1.

Empty columns indicate errors.

**Assignment statements****IncrementAssign (+=)****(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  $\alpha$  and  $\beta$  are added and the result is assigned to  $\alpha$ .

**<2> When there is a register specification**

$\alpha$  is assigned to the specified register.

The contents of the specified register are added to  $\beta$  and their result is assigned to the specified register.

The contents of the specified register are assigned to  $\alpha$ .

**<3> Increment with carry; no register specification**

An increment with carry operation is performed using the two operands  $\alpha$  and  $\beta$ , and the result is assigned to  $\alpha$ .

**<4> Increment with carry; with register specification**

The contents of  $\alpha$  are assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

**[Description]****<1> When there is no register specification**

The contents of  $\alpha$  and  $\beta$  can be entered in ADD, ADDW, ADDG, ADDWG.

**<2> When there is a register specification**

The contents of  $\alpha$  can be entered in MOV, MOVW and MOVG.

The contents of  $\beta$  can be entered in ADD, ADDW, ADDG, ADDWG.

**<3> Increment with carry; no register specification**

The contents of  $\alpha$  and  $\beta$  can be entered in ADDC.

**<4> Increment with carry; with register specification**

The contents of  $\alpha$  can be entered in MOV.

The contents of  $\beta$  can be entered in ADDC.

Assignment statements	IncrementAssign (+=)
<b>[Generated instructions]</b>	
<b>&lt;1&gt; When there is no register specification</b>	
ADD	$\alpha, \beta$
ADDW, ADDG, and ADDWG may be generated instead, depending on the operand.	
<b>&lt;2&gt; When there is a register specification</b>	
MOV	specified register, $\alpha$
ADD	specified register, $\beta$
MOV	$\alpha$ , specified register
ADDW, ADDG, and ADDWG may be generated instead, depending on the operand.	
<b>&lt;3&gt; Increment with carry; no register specification</b>	
ADDC	$\alpha, \beta$
<b>&lt;4&gt; Increment with carry; with register specification</b>	
MOV	specified register, $\alpha$
ADDC	specified register, $\beta$
MOV	$\alpha$ , specified register
For details of combinations of $\alpha$ and $\beta$ , see <b>Table 4-6. Generated Instructions for Increment Assignments</b> . Depending on the entered statement, $\alpha$ indicates the specified register.	

## Assignment statements

## IncrementAssign (+=)

## [Use examples]

## &lt;1&gt; When there is no register specification

```

ADD      X, #0C0H      ; X+=#0C0H
ADDW     BC, #0C00H    ; BC+=#0C00H
ADDG     VVP, UUP      ; VVP+=UUP
ADDWG    SP, #4000H    ; SP+=#4000H

```

## &lt;2&gt; 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
MOVG     WHL, VVP      ; VVP+=DDL (WHL)
ADDG     WHL, DDLG
MOVG     VVP, WHL

```

## &lt;3&gt; Increment with carry; no register specification

```

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

```

## &lt;4&gt; Increment with carry; with register specification

```

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

```

Assignment statements

IncrementAssign (+=)

Table 4-6. Generated Instructions for Increment Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY																					
	b	Bit symbol																					
	c	Byte user symbol			*1	*1	*1	*1				*2	*2	*2						*1			*1
	d	Byte data			*1	*1	*1	*1			*1									*1			*1
	e	A			*1	*1	*1	*1	*1		*1									*1	*1	*1	*1
	f	Byte register			*1	*1	*1	*1	*1		*1									*1			*1
	g	sfr					*1	*1															*1
	h	Special register																					
	i	Word user symbol	*4			*1	*1	*1				*2	*2	*2	*2					*7			*2
	j	Word data			*2							*2	*2	*2	*2					*2			*2
	k	AX			*2							*2	*2	*2	*2	*2				*2			*2
	l	Word register			*2							*2	*2	*2	*2	*2				*2			*2
	m	sfrp												*2	*2								*2
	n	Triple byte specification																					
	o	WHL			*3							*3					*3	*3	*3	*3			*3
	p	Triple byte register																*3	*3				*3
q	SP																					*4	
r	Constant			*1	*1	*1	*1				*2	*2	*2	*2					*1			*1	
s	Direct access symbol					*1							*2										
t	Indirect access symbol					*1																	
u	Immediate symbol																						

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

\*2 : Generates ADDW instruction.

\*3 : Generates ADDG instruction.

\*4 : Generates ADDWG instruction.

Empty spaces indicate errors.



## Assignment statements

## DecrementAssign (--)

## (3) DecrementAssign (--)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

$\beta$  is subtracted from  $\alpha$  and the result is assigned to  $\alpha$ .

## &lt;2&gt; When there is a register specification

$\alpha$  is assigned to the specified register.

$\beta$  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  $\alpha$ .

## &lt;3&gt; Decrement with carry; no register specification

A decrement with carry operation is performed using the two operands  $\alpha$  and  $\beta$ , and the result is assigned to  $\alpha$ .

## &lt;4&gt; Decrement with carry; with register specification

The contents of  $\alpha$  are assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; When there is no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in SUB, SUBW, SUBG, and SUBWG.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  can be entered in MOV, MOVW, and MOVG.

The contents of  $\beta$  can be entered in SUB, SUBW, SUBG, and SUBWG.

## &lt;3&gt; Decrement with carry; no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in SUBC.

## &lt;4&gt; Decrement with carry; with register specification

The contents of  $\alpha$  can be entered in MOV.

The contents of  $\beta$  can be entered in SUBC.

## Assignment statements

## DecrementAssign (--)

**[Generated instructions]****<1> When there is no register specification**

The following instruction is generated.

SUB             $\alpha, \beta$

SUBW, SUBG, and SUBWG may be generated instead, depending on the operand.

**<2> When there is a register specification**

The following instruction is generated.

MOV            specified register,  $\alpha$

SUB            specified register,  $\beta$

MOV             $\alpha$ , specified register

SUBW, SUBG, and SUBWG may be generated instead, depending on the operand.

**<3> Decrement with carry; no register specification**

The following instruction is generated.

SUBC             $\alpha, \beta$

**<4> Decrement with carry; with register specification**

The following instruction is generated.

MOV            specified register,  $\alpha$

SUBC            specified register,  $\beta$

MOV             $\alpha$ , specified register

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

## Assignment statements

## DecrementAssign (--)

## [Use examples]

## &lt;1&gt; When there is no register specification

```

SUB      X, #0C0H      ; X--=#0C0H
SUBW     BC, #0C00H     ; BC--=#0C00H
SUBG     VVP, UUP       ; VVP-=UUP
SUBWG    SP, #4000H     ; SP--=#4000H

```

## &lt;2&gt; 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
MOVG     WHL, VVP       ; VVP-=DDL (WHL)
SUBG     WHL, DDL
MOVG     VVP, WHL

```

## &lt;3&gt; Decrement with carry; no register specification

```

SUBC     A, #50H        ; A--=#50H, CY

```

## &lt;4&gt; Decrement with carry; with register specification

```

MOV      A, PSWH        ; PSWH--=#50H, CY (A)
SUBC     A, #50H
MOV      PSWH, A

```

Assignment statements

DecrementAssign (--)

Table 4-7. Generated Instructions for Decrement Assignments

			$\beta$																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	
$\alpha$	a	CY																						
	b	Bit symbol																						
	c	Byte user symbol			*1	*1	*1	*1				*2	*2	*2						*1			*1	
	d	Byte data			*1	*1	*1	*1			*1									*1			*1	
	e	A			*1	*1	*1	*1	*1		*1									*1	*1	*1	*1	
	f	Byte register			*1	*1	*1	*1	*1		*1									*1			*1	
	g	sfr					*1	*1															*1	
	h	Special register																						
	i	Word user symbol				*1	*1	*1				*2	*2	*2	*2						*7			*2
	j	Word data			*2							*2	*2	*2	*2						*2			*2
	k	AX			*2							*2	*2	*2	*2	*2					*2			*2
	l	Word register			*2							*2	*2	*2	*2	*2					*2			*2
	m	sfrp												*2	*2									*2
	n	Triple byte specification																						
	o	WHL			*3							*3					*3	*3	*3		*3			*3
	p	Triple byte register																*3	*3					*3
	q	SP																						*4
r	Constant			*1	*1	*1	*1				*2	*2	*2	*2						*1			*1	
s	Direct access symbol					*1							*2											
t	Indirect access symbol					*1																		
u	Immediate symbol																							

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

\*2 : Generates SUBW instruction.

\*3 : Generates SUBG instruction.

\*4 : Generates SUBWG instruction.

Empty spaces indicate errors.

## Assignment statements

## MultiplyAssign (\*=)

## (4) MultiplyAssign (\*=)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

$\alpha$  and  $\beta$  are multiplied, and the result is assigned to  $\alpha$ .

## &lt;2&gt; When there is a register specification

$\alpha$  is assigned to the specified register.

The specified register is multiplied with  $\beta$ , and the result is assigned to the specified register.

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; Where there is no register specification

The contents of  $\alpha$  can be entered only in AX.

The contents of  $\beta$  can be entered in MULU and MULUW.

## &lt;2&gt; Where there is a register specification

The specified register can be entered only in AX.

The contents of  $\alpha$  can be entered in MOVW.

The contents of  $\beta$  can be entered in MULU and MULUW.

## [Generated instructions]

## &lt;1&gt; When there is no register specification

MULU             $\beta$

However, MULUW may be generated depending on the operand.

## &lt;2&gt; When there is a register specification

MOVW            AX,  $\alpha$

MULU             $\beta$

MOVW             $\alpha$ , AX

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-8. Generated Instructions for Multiply Assignments.**

---

**Assignment statements****MultiplyAssign (\*=)**

---

**[Use examples]****<1> When there is no register specification**

```
MULU      B           ; AX*=B
MULUW     BC          ; AX*=BC
```

**<2> When there is a register specification**

```
MOVW      AX, DAT      ; DAT*=C (AX)
MULU      C
MOVW      DAT, AX
MOVW      AX, AATP      ; AATP*=DE (AX)
MULUW     DE
MOVW      AATP, AX
```

Assignment statements

MultiplyAssign (\*=)

Table 4-8. Generated Instructions for Multiply Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY																					
	b	Bit symbol																					
	c	Byte user symbol																					
	d	Byte data																					
	e	A																					
	f	Byte register																					
	g	sfr																					
	h	Special register																					
	i	Word user symbol																					
	j	Word data																					
	k	AX					*1	*1					*2	*2									
	l	Word register																					
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
	q	SP																					
	r	Constant																					
	s	Direct access symbol																					
	t	Indirect access symbol																					
	u	Immediate symbol																					

\*1 : Generates Mulu instructions.

\*2 : Generates MULUW instructions.

Empty columns indicate errors.

Assignment statements	DivideAssign (/=)
-----------------------	-------------------

(5) DivideAssign (/=)

[Coding format]

$[\Delta] \text{ [size specification] } [\Delta] \alpha \text{ } [\Delta] \text{ } /= \text{ } [\Delta] \beta$
--

[Function]

$\alpha$  is divided by  $\beta$ , and the result is assigned to  $\alpha$ .

[Description]

The contents of  $\alpha$  can be entered only in AX.  
The contents of  $\beta$  can be entered in DIVUW and DIVUX.

[Generated instructions]

DIVUW      C  
However, DIVUX may be generated depending on the operand.

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-9. Generated Instructions for Divide Assignments.**

[Use examples]

DIVUW	B	; AX /= B
DIVUX	BC	; AX /= BC



Assignment statements

DivideAssign (/=)

Table 4-9. Generated Instructions for Divide Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY																					
	b	Bit symbol																					
	c	Byte user symbol																					
	d	Byte data																					
	e	A																					
	f	Byte register																					
	g	sfr																					
	h	Special register																					
	i	Word user symbol																					
	j	Word data																					
	k	AX					*1	*1					*2	*2									
	l	Word register																					
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
	q	SP																					
	r	Constant																					
	s	Direct access symbol																					
	t	Indirect access symbol																					
	u	Immediate symbol																					

\*1 : Generates DIVUW

\*2 : Generates DIVUX

Empty columns indicate errors.

## Assignment statements

## LogicalANDAssign (&amp;=)

## (6) LogicalANDAssign (&amp;=)

## [Coding format]

$[\Delta]$ [size specification] $[\Delta]$ $\alpha$ $[\Delta]$ $\&=$ $[\Delta]$ [size specification] $[\Delta]$ $\beta$
---

## [Function]

## &lt;1&gt; When there is no register specification

The logical AND ( $\alpha \& \beta$ ) is obtained from the bits in  $\alpha$  and  $\beta$ , and the result is assigned to  $\alpha$ .

## &lt;2&gt; When there is a register specification

$\alpha$  is assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; Where there is no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in AND or AND1.

## &lt;2&gt; Where there is a register specification

The contents of  $\alpha$  can be entered in MOV or MOV1.

The contents of  $\beta$  can be entered in AND or AND1.

## [Generated instructions]

## &lt;1&gt; When there is no register specification

• When  $\alpha$  is CY

AND1      CY,  $\beta$

• When  $\alpha$  is not CY

AND       $\alpha$ ,  $\beta$

## Assignment statements

## LogicalANDAssign (&amp;=)

## &lt;2&gt; When there is a register specification

## • When the specified register is CY

MOV1 CY,  $\alpha$ AND1 CY,  $\beta$ MOV1  $\alpha$ , CY

## • When the specified register is not CY

MOV specified register,  $\alpha$ AND specified register,  $\beta$ MOV  $\alpha$ , specified register

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-10. Generated Instructions for Logical AND Assignments.**

## [Use examples]

## &lt;1&gt; When there is no register specification

AND1 CY, P1S.1 ; CY&amp;=P1S.1

AND A, #0FFH ; A&amp;=#0FFH

## &lt;2&gt; When there is a register specification

MOV1 CY, A.1 ; A.1&amp;=PORT3.0 (CY)

AND1 CY, PORT3.0

MOV1 A.1, CY

MOV A, [DE] ; [DE] &amp;=#07H (A)

AND A, #07H

MOV [DE], A

Assignment statements

LogicalANDAssign (&amp;=)

Table 4-10. Generated Instructions for Logical AND Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY		*2	*2					*2													
	b	Bit symbol																					
	c	Byte user symbol			*1	*1	*1	*1												*1			*1
	d	Byte data			*1	*1	*1	*1			*1									*1			*1
	e	A			*1	*1	*1	*1	*1		*1									*1	*1	*1	*1
	f	Byte register			*1	*1	*1	*1	*1		*1									*1			*1
	g	sfr					*1	*1															*1
	h	Special register																					
	i	Word user symbol																					
	j	Word data																					
	k	AX																					
	l	Word register																					
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
q	SP																						
r	Constant			*1	*1	*1	*1												*1			*1	
s	Direct access symbol					*1																	
t	Indirect access symbol					*1																	
u	Immediate symbol																						

\*1 : Generates AND instruction.

\*2 : Generates AND1 instruction.

Empty spaces indicate errors.

## Assignment statements

## LogicalORAssign (|=)

## (7) LogicalORAssign (|=)

## [Coding format]

$[\Delta] \text{ [size specification] } [\Delta] \alpha [\Delta]  = [\Delta] \text{ [size specification] } [\Delta] \beta$
--

## [Function]

## &lt;1&gt; When there is no register specification

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

## &lt;2&gt; When there is a register specification

$\alpha$  is assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; When there is no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in OR or OR1.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  can be entered in MOV or MOV1.

The contents of  $\beta$  can be entered in OR or OR1.

## [Generated instructions]

## &lt;1&gt; When there is no register specification

• When  $\alpha$  is CY

OR1      CY,  $\beta$

• When  $\alpha$  is not CY

OR       $\alpha$ ,  $\beta$

## Assignment statements

## LogicalORAssign (|=)

## &lt;2&gt; When there is a register specification

- When the specified register is CY

```
MOV1    CY,  $\alpha$ 
```

```
OR1     CY,  $\beta$ 
```

```
MOV1     $\alpha$ , CY
```

- When the specified register is not CY

```
MOV     specified register,  $\alpha$ 
```

```
OR      specified register,  $\beta$ 
```

```
MOV      $\alpha$ , specified register
```

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-11. Generated Instructions for Logical OR Assignments.**

## [Use examples]

## &lt;1&gt; When there is no register specification

```
OR1     CY, P1S.1      ;CY |= P1S.1
```

```
OR      A, #0FFH       ;A |= #0FFH
```

## &lt;2&gt; When there is a register specification

```
MOV1    CY, A.1        ;A.1 |= PORT3.0 (CY)
```

```
OR1     CY, PORT3.0
```

```
MOV1    A.1, CY
```

```
MOV     A, [DE]         ;[DE] |= #07H (A)
```

```
OR      A, #07H
```

```
MOV     [DE], A
```

Assignment statements

LogicalORAssign (|=)

Table 4-11. Generated Instructions for Logical OR Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY		*2	*2					*2													
	b	Bit symbol																					
	c	Byte user symbol			*1	*1	*1	*1												*1			*1
	d	Byte data			*1	*1	*1	*1			*1									*1			*1
	e	A			*1	*1	*1	*1	*1		*1									*1	*1	*1	*1
	f	Byte register			*1	*1	*1	*1	*1		*1									*1			*1
	g	sfr					*1	*1															*1
	h	Special register																					
	i	Word user symbol																					
	j	Word data																					
	k	AX																					
	l	Word register																					
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
	q	SP																					
	r	Constant			*1	*1	*1	*1												*1			*1
	s	Direct access symbol					*1																
	t	Indirect access symbol					*1																
	u	Immediate symbol																					

\*1 : Generates OR instruction.

\*2 : Generates OR1 instruction.

Empty spaces indicate errors.

## Assignment statements

LogicalXORAssign ( $\wedge=$ )(8) LogicalXORAssign ( $\wedge=$ )

## [Coding format]

$[\Delta]$ [size specification] $[\Delta]$ $\alpha$ $[\Delta]$ $\wedge=$ $[\Delta]$ [size specification] $[\Delta]$ $\beta$
---

## [Function]

## &lt;1&gt; When there is no register specification

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

## &lt;2&gt; When there is a register specification

$\alpha$  is assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; When there is no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in XOR or XOR1.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  can be entered in MOV or MOV1.

The contents of  $\beta$  can be entered in XOR or XOR1.

## [Generated instructions]

## &lt;1&gt; When there is no register specification

- When  $\alpha$  is CY  
XOR1      CY,  $\beta$
- When  $\alpha$  is not CY  
XOR       $\alpha$ ,  $\beta$



## Assignment statements

LogicalXORAssign ( $\wedge=$ )

## &lt;2&gt; When there is a register specification

## • When the specified register is CY

MOV1 CY,  $\alpha$ XOR1 CY,  $\beta$ MOV1  $\alpha$ , CY

## • When the specified register is not CY

MOV specified register,  $\alpha$ XOR specified register,  $\beta$ MOV  $\alpha$ , specified register

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-12. Generated Instructions for Logical XOR Assignments.**

## [Use examples]

## &lt;1&gt; When there is no register specification

XOR1 CY, P1S.1 ; CY $\wedge$ =P1S.1XOR A, #0FFH ; A $\wedge$ =#0FFH

## &lt;2&gt; When there is a register specification

MOV1 CY, A.1 ; A.1 $\wedge$ =PORT3.0 (CY)

XOR1 CY, PORT3.0

MOV1 A.1, CY

MOV A, [DE] ; [DE] $\wedge$ =#07H (A)

XOR A, #07H

MOV [DE], A

Assignment statements

LogicalXORAssign ( $\wedge=$ )

Table 4-12. Generated Instructions for Logical XOR Assignments

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY		*2	*2					*2													
	b	Bit symbol																					
	c	Byte user symbol			*1	*1	*1	*1												*1			*1
	d	Byte data			*1	*1	*1	*1			*1									*1			*1
	e	A			*1	*1	*1	*1	*1		*1									*1	*1	*1	*1
	f	Byte register			*1	*1	*1	*1	*1		*1									*1			*1
	g	sfr					*1	*1															*1
	h	Special register																					
	i	Word user symbol																					
	j	Word data																					
	k	AX																					
	l	Word register																					
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
q	SP																						
r	Constant			*1	*1	*1	*1												*1			*1	
s	Direct access symbol					*1																	
t	Indirect access symbol					*1																	
u	Immediate symbol																						

\*1 : Generates XOR instruction.

\*2 : Generates XOR1 instruction.

Empty spaces indicate errors.

**Assignment statements****RightShiftAssign (>>=)****(9) RightShiftAssign (>>=)****[Coding format]**

$[\Delta]$ [size specification] $[\Delta]$ $\alpha$ $[\Delta]$ $>>=$ $[\Delta]$ $\beta$ $[\Delta]$ [register specification]
---

**[Function]****<1> When there is no register specification**

$\alpha$  is shifted to the right of the  $\beta$  bit, and the result is assigned to  $\alpha$ .

**<2> When there is a register specification**

$\alpha$  is assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

**[Description]****<1> When there is no register specification**

The contents of  $\alpha$  and  $\beta$  can be entered in SHR and SHRW.

**<2> When there is a register specification**

The contents of  $\alpha$  can be entered in MOV and MOVW.

The contents of  $\beta$  can be entered in SHR and SHRW.

**[Generated instructions]****<1> When there is no register specification**

SHR  $\alpha, \beta$

However, SHRW may be generated depending on the operand.

**<2> When there is a register specification**

MOV specified register,  $\alpha$

SHR specified register,  $\beta$

MOV  $\alpha$ , specified register

However, SHRW may be generated depending on the operand.

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-13. Generated Instructions for Right Shift Assignments.**

Use  $\alpha$  to indicate the specified register.

---

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

---

**[Use examples]****<1> When there is no register specification**

```
SHR      A, 4           ; A>>=4
SHRW     AX, 8          ; AX>>=8
```

**<2> When there is a register specification**

```
MOV      A, CCV          ; CCV>>=4 (A)
SHR      A, 4
MOV      CCV, A
MOVW     AX, RSRP        ; RSRP>>=12 (AX)
SHRW     AX, 12
MOVW     RSRP, AX
```

Assignment statements

RightShiftAssign (&gt;&gt;=)

Table 4-13. Generated Instructions for RightShiftAssign

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY																					
	b	Bit symbol																					
	c	Byte user symbol																					
	d	Byte data																					
	e	A			*1					*1										*1			
	f	Byte register			*1					*1										*1			
	g	sfr																					
	h	Special register																					
	i	Word user symbol																					
	j	Word data																					
	k	AX			*2					*2										*2			
	l	Word register			*2					*2										*2			
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
	q	SP																					
	r	Constant																					
	s	Direct access symbol																					
	t	Indirect access symbol																					
	u	Immediate symbol																					

\*1 : Generates SHR instruction.

\*2 : Generates SHRW instruction.

Empty spaces indicate errors.

## Assignment statements

## LeftShiftAssign (&lt;&lt;=)

## (10) LeftShiftAssign (&lt;&lt;=)

## [Coding format]

$[\Delta]$ [size specification] $[\Delta]$ $\alpha$ $[\Delta]$ <<= $[\Delta]$ $\beta$ $[\Delta]$ [register specification]
---

## [Function]

## &lt;1&gt; When there is no register specification

$\alpha$  is shifted to the left of the  $\beta$  bit, and the result is assigned to  $\alpha$ .

## &lt;2&gt; When there is a register specification

$\alpha$  is assigned to the specified register.

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

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; When there is no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in SHL and SHLW.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  can be entered in MOV and MOVW.

The contents of  $\beta$  can be entered in SHL and SHLW.

## [Generated instructions]

## &lt;1&gt; When there is no register specification

SHL  $\alpha, \beta$

However, SHLW is generated depending on the operand.

## &lt;2&gt; When there is a register specification

MOV specified register,  $\alpha$

SHL specified register,  $\beta$

MOV  $\alpha$ , specified register

However, SHLW may be generated depending on the operand.

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-14. Generated Instructions for Left Shift Assignments.**

Use  $\alpha$  to indicate the specified register.

---

**Assignment statements****LeftShiftAssign (<<=)**

---

**[Use examples]****<1> When there is no register specification**

```
SHL      A, 4           ; A<<=4
SHLW     AX, 8          ; AX<<=8
```

**<2> When there is a register specification**

```
MOV      A, CCV          ; CCV<<=4 (A)
SHL      A, 4
MOV      CCV, A
MOVW     AX, RSRP        ; RSRP<<=12 (AX)
SHLW     AX, 12
MOVW     RSRP, AX
```

Assignment statements

LeftShiftAssign (&gt;&gt;=)

Table 4-14. Generated Instructions for LeftShiftAssign

			$\beta$																					
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	
$\alpha$	a	CY																						
	b	Bit symbol																						
	c	Byte user symbol																						
	d	Byte data																						
	e	A			*1					*1										*1				
	f	Byte register			*1					*1										*1				
	g	sfr																						
	h	Special register																						
	i	Word user symbol																						
	j	Word data																						
	k	AX			*2					*2										*2				
	l	Word register			*2					*2										*2				
	m	sfrp																						
	n	Triple byte specification																						
	o	WHL																						
	p	Triple byte register																						
	q	SP																						
	r	Constant																						
	s	Direct access symbol																						
	t	Indirect access symbol																						
	u	Immediate symbol																						

\*1 : Generates SHL instruction.

\*2 : Generates SHLW instruction.

Empty spaces indicate errors.



Count statements	Increment (++)
------------------	----------------

(11) Increment (++)

[Coding format]

[ $\Delta$ ] [size specification] [ $\Delta$ ]  $\alpha$  [ $\Delta$ ] ++

[Function]

1 is added to the contents of  $\alpha$ .

[Description]

The contents of  $\alpha$  can be entered in INC, INCW or INCG.

[Generated instructions]

INCW         $\alpha$   
INCW or INCG may be generated depending on the operands.  
For details of  $\alpha$ , see **Table 4-15. Generated Instructions for Increment.**

[Use examples]

INC	H	;H++
INC	CNT	;CNT++
INCW	HL	;HL++
INCW	SSP	;SSP++
INCG	WHL	;WHL++
INCG	SP	;SP++

Count statements

Increment (++)

Table 4-15. Generated Instructions for Increment

$\alpha$	a	CY	
	b	Bit symbol	
	c	Byte user symbol	*1
	d	Byte data	*1
	e	A	*1
	f	Byte register	*1
	g	sfr	
	h	Special register	
	i	Word user symbol	*2
	j	Word data	*2
	k	AX	*2
	l	Word register	*2
	m	sfrp	
	n	Triple byte specification	
	o	WHL	*3
	p	Triple byte register	*3
	q	SP	*3
	r	Constant	*1
	s	Direct access symbol	
	t	Indirect access symbol	
	u	Immediate symbol	

\*1 : Generates INC instruction.

\*2 : Generates INCW instruction.

\*3 : Generates INCG instruction.

Empty spaces indicate errors.

Count statements	Decrement (– –)
------------------	-----------------

(12) 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, DECW, or DECG.

[Generated instructions]

DEC            α  
DECW or DECG may be generated depending on the operands.  
For details of α, see **Table 4-16. Generated Instructions for Decrement.**

[Use examples]

DEC	H	; H--
DEC	CNT	; CNT--
DECW	HL	; HL--
DECW	SSP	; SSP--
DECG	WHL	; WHL--
DECG	SP	; SP--

Count statements

Decrement (– –)

Table 4-16. Generated Instructions for Decrement

$\alpha$	a	CY	
	b	Bit symbol	
	c	Byte user symbol	*1
	d	Byte data	*1
	e	A	*1
	f	Byte register	*1
	g	sfr	
	h	Special register	
	i	Word user symbol	*2
	j	Word data	*2
	k	AX	*2
	l	Word register	*2
	m	sfrp	
	n	Triple byte specification	
	o	WHL	*3
	p	Triple byte register	*3
	q	SP	*3
	r	Constant	*1
	s	Direct access symbol	
	t	Indirect access symbol	
	u	Immediate symbol	

\*1 : Generates DEC instruction.

\*2 : Generates DECW instruction.

\*3 : Generates DECG instruction.

Empty spaces indicate errors.

## Exchange statements

## Exchange (&lt;-&gt;)

## (13) Exchange (&lt;-&gt;)

## [Coding format]

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

## [Function]

## &lt;1&gt; When there is no register specification

The contents of  $\alpha$  and  $\beta$  are exchanged.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  are assigned to the specified register.

The contents of the specified register are exchanged with the contents of  $\beta$ .

The contents of the specified register are assigned to  $\alpha$ .

## [Description]

## &lt;1&gt; Where there is no register specification

The contents of  $\alpha$  and  $\beta$  can be entered in XCH or XCHW.

## &lt;2&gt; When there is a register specification

The contents of  $\alpha$  can be entered in MOV and MOVW.

The contents of  $\beta$  can be entered in XCH and XCHW.

## [Generated instructions]

## &lt;1&gt; When there is no register specification

XCH             $\alpha, \beta$

XCHW may be generated depending on the operands.

## &lt;2&gt; When there is a register specification

MOV            specified register,  $\alpha$

XCH            specified register,  $\beta$

MOV             $\alpha$ , specified register

XCHW may be generated depending on the operands.

For details of combinations of  $\alpha$  and  $\beta$ , see **Table 4-17. Generated Instructions for Exchange.**

$\alpha$  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 (&lt;-&gt;)

Table 4-17. Generated Instructions for Exchange

			$\beta$																				
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
$\alpha$	a	CY																					
	b	Bit symbol																					
	c	Byte user symbol			*1	*1					*2									*1			
	d	Byte data			*1	*1				*1										*1			
	e	A			*1	*1	*1	*1	*1	*1										*1	*1	*1	
	f	Byte register			*1	*1	*1	*1	*1	*1										*1	*1		
	g	sfr																					
	h	Special register																					
	i	Word user symbol				*1					*2	*2								*2			
	j	Word data			*2						*2	*2								*2			
	k	AX			*2						*2	*2	*2	*2	*2					*2	*2	*2	
	l	Word register			*2						*2	*2	*2	*2	*2					*2	*2		
	m	sfrp																					
	n	Triple byte specification																					
	o	WHL																					
	p	Triple byte register																					
	q	SP																					
	r	Constant			*1	*1					*2	*2								*1			
	s	Direct access symbol																					
	t	Indirect access symbol																					
	u	Immediate symbol																					

\*1 : Generates XCH instructions.

\*2 : Generates XCHW instructions.

Empty spaces indicate errors.

Bit manipulation statements

Set bit (=)

(15) Set bit (=)

[Coding format]

$[\Delta] \ \alpha 1 \ [\Delta] \ [= [\Delta] \ \alpha 2 \ [\Delta] \cdots] = [\Delta] \ 1 \ [\Delta] \ [(CY \text{ specification})]$   
 Enter "1" at the end of the right side.

[Function]

<1> When there is no CY specification

$\alpha n$  is set (to a value of "1").

<2> When there is a CY specification

CY and  $\alpha n$  are set (to a value of "1").

[Description]

The contents of  $\alpha n$  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 33 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             $\alpha 1$

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

SET1             $\alpha n$

SET1             $\alpha n-1$

:

SET1             $\alpha 2$

SET1             $\alpha 1$

<3> When there is a CY specification

SET1            CY

SET1             $\alpha 1$



Bit manipulation statements	Set bit (=)
<b>&lt;4&gt; When there is a CY specification in sequential assignments</b>	
SET1          CY	
SET1 $\alpha n$	
SET1 $\alpha n-1$	
:	
SET1 $\alpha 2$	
SET1 $\alpha 1$	
For details, see <b>Table 4-18. Generated Instructions for Set Bit</b>	
<b>[Use examples]</b>	
<b>&lt;1&gt; When there is no CY specification</b>	
SET1          A.3	;A.3=1
SET1          CY	;CY=1
SET1          BIT3	;BIT1=BIT2=BIT3=1
SET1          BIT2	
SET1          BIT1	
<b>&lt;2&gt; When there is a CY specification</b>	
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 4-18. Generated Instructions for Set Bit

$\alpha$	a	CY	*1
	b	Bit symbol	*1
	c	Byte user symbol	*1
	d	Byte data	
	e	A	
	f	Byte register	
	g	sfr	
	h	Special register	
	i	Word user symbol	*1
	j	Word data	
	k	AX	
	l	Word register	
	m	sfrp	
	n	Triple byte specification	
	o	WHL	
	p	Triple byte register	
	q	SP	
	r	Constant	
	s	Direct access symbol	
	t	Indirect access symbol	
	u	Immediate symbol	

\*1 : Generates SET1 instruction.

Empty spaces indicate errors.

## Bit manipulation statements

## Clear bit (=)

## (16) 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]

## &lt;1&gt; When there is no CY specification

$\alpha n$  is cleared (to a value of "0").

## &lt;2&gt; When there is a CY specification

CY and  $\alpha n$  are cleared (to a value of "0").

## [Description]

The contents of  $\alpha n$  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 33 are entered. If even one error occurs during sequential assignments, no instructions will be generated.

## [Generated instructions]

## &lt;1&gt; When there is no CY specification

```
CLR1      α 1
```

## &lt;2&gt; When there is no CY specification in sequential assignments

```
CLR1      α n
```

```
CLR1      α n-1
```

```
:
```

```
CLR1      α 2
```

```
CLR1      α 1
```

## &lt;3&gt; When there is a CY specification

```
CLR1      CY
```

```
CLR1      α 1
```

## Bit manipulation statements

## Clear bit (=)

## &lt;4&gt; When there is a CY specification in sequential assignments

```

CLR1      CY
CLR1      α n
CLR1      α n-1
:
CLR1      α 2
CLR1      α 1

```

For details, see **Table 4-19. Generated Instructions for Clear Bit**

## [Use examples]

## &lt;1&gt; 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

```

## &lt;2&gt; 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

```

Bit manipulation statements

Clear bit (=)

Table 4-19. Generated Instructions for Clear Bit

$\alpha$	a	CY	*1
	b	Bit symbol	*1
	c	Byte user symbol	*1
	d	Byte data	
	e	A	
	f	Byte register	
	g	sfr	
	h	Special register	
	i	Word user symbol	*1
	j	Word data	
	k	AX	
	l	Word register	
	m	sfrp	
	n	Triple byte specification	
	o	WHL	
	p	Triple byte register	
	q	SP	
	r	Constant	
	s	Direct access symbol	
	t	Indirect access symbol	
	u	Immediate symbol	

\*1 : Generates CLR1 instruction.

Empty spaces indicate errors.

[MEMO]

## CHAPTER 5 DIRECTIVES

### 5.1 Overview of Directives

Directives are entered into source programs as various directives that the ST78K4 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.

#DEFINE

#define

#DEFINE

---

**(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.



#DEFINE

#define

#DEFINE

---

**[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
CMP     X,#1      ; if( X == #1 )
BNZ     $?L1
MOV     A,#0C5H   ; A = #0C5H
?L1:                                ; endif
```

#IFDEF/#ELSE/#ENDIF

#ifndef/#else/#endif

#IFDEF/#ELSE/#ENDIF

---

**(2) Conditional processing directive (#ifndef/#else/#endif)****[Coding format]**

```
[Δ] # [Δ] ifndef 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>st78k4 -c4026 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>st78k4 -c4026 sample.st
```

**<Output source program>**

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

#INCLUDE

#include

#INCLUDE

---

**(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 ST78K4 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 ST78K4 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
```

#DEFCALLT

#defcallt

#DEFCALLT

**(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

### 6.1 Overview of Control Instructions

Control instructions, which are entered into the source program, set various directives that the ST78K4 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 (\$KANJI CODE). For description of the coding format for other control instructions, see the “**RA78K4 Series Assembler Package User’s Manual 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] NODEBAG
[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] CHGSFR [Δ] ( [Δ] constant [Δ] )
[Δ] \$ [Δ] CHGSFRA
[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] ) or [Δ] \$ [Δ] _IF Δ conditional expression
[Δ] \$ [Δ] ELSEIF [Δ] ( [Δ] switch name [ [Δ] : [Δ] switch name... [Δ] ) or [Δ] \$ [Δ] _ELSEIF 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	\$KANJI CODE

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

**\$PROCESSOR****\$processor****\$PROCESSOR**

---

**(1) Processor type specification instruction (\$PROCESSOR)****[Coding format]**

<pre>[Δ] \$ [Δ] PROCESSOR [Δ] ( [Δ] model name [Δ] ) Abbreviated form: [Δ] \$ [Δ] PC [Δ] ( [Δ] model name [Δ] )</pre>
---

**[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 (4021)
```

**\$SYMLEN/\$NOSYMLEN****\$symlen/\$nosymlen****\$SYMLEN/\$NOSYMLEN**

---

**(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
```

**\$KANJI CODE****\$kanjicode****\$KANJI CODE****(3) Kanji code specification control instruction (\$KANJI CODE)****[Coding format]**

[Δ] \$ [Δ] KANJI CODE 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 : \$KANJI CODE SJIS  
 EUC : \$KANJI CODE EUC  
 NONE : \$KANJI CODE 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]**

```
$KANJI CODE SJIS
```

[MEMO]

## CHAPTER 7 PRODUCT OVERVIEW

### 7.1 Product Contents

The files listed below in “Table 7-1. Bundled Files” are provided with the ST78K4 structured assembler. Accordingly, these files are also bundled with the RA78K4 assembler package.

**Table 7-1. Bundled Files**

File name	Description
st78k4.exe st78k4.hlp	Structured assembler (executable) Structured assembler help file
test1.s test2.s testinc.s	Sample programs
st.bat	Batch files

Executable : Executable files are the first files to be read into memory when a program is launched.

Sample programs : Sample programs are files that are used to check the operation of the structured assembler.

### 7.2 System Configuration

Host machines and OSs supported by the ST78K4 are the same as those supported by the RA78K4 assembler package. For details, see “**RA78K4 Assembler Package Operation**”.

### 7.3 Installation

The installation method for the ST78K4 is the same as for the RA78K4 assembler package. For details, see “**RA78K4 Assembler Package Operation**”.

[MEMO]



## CHAPTER 8 OPERATION METHODS

### 8.1 Input/Output File Types

The structured assembler's input/output files are listed below in Table 8-1. Structured Assembler's Input/Output Files.

**Table 8-1. Structured Assembler's Input/Output Files**

	File type	Default file type
Input file	Source module files These are source module files that are coded in structured assembly language.	None
	Parameter files (Note) These files contain options that enable structured assembler options to be specified from files.	.PST
Output files	Secondary source files These are source files that are coded in assembly language.	.ASM
	Error list files These files contain the structured assembler's error data.	.EST

**Caution :** See “8.3.1 Launch via parameter file”.

### 8.2 Option Functions

Option functions are included to facilitate use of the structured assembler. The option functions can be specified via the structured assembler command line or via a parameter file. For details of specific option functions, see “8.4 Options”.



**Example 1.** Parameter file “ST.JOB”, created using a text editor.

- Contents of ST.JOB

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

- Specification of “ST.JOB” to launch the structured assembler.

```
A>ST78K4 -FST.JOB
Structured assembler preprocessor for RA78K/IV Vx.xx [xx xxx xx]
  Copyright (C) NEC Corporation 19xx,xxxx

start

Target chip : uPD784xxx
Device file : Vx.xx

A>
```

**Example 2.** Command line revision or augmentation of parameter file contents

```
A>ST78K4 -FST.JOB -OB:
A>
```

### 8.3.2 Execution start and end messages

#### (1) Execution start message

When the structured assembler is launched, the following execution start message is output to the console.

```
Structured assembler preprocessor for RA78K/IV Vx.xx [xx xxx xx]  
Copyright (C) NEC Corporation 19xx,xxxx
```

#### (2) Processing progress message

```
start...
```

Another dot is added after “start” each time 100 lines of code have been executed.

#### (3) Execution end message

When execution ends with no errors detected, the following message is output to the console and control is returned to the OS.

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

When errors are detected, the number of errors is indicated in the following message, which is output to the console before control is returned to the OS.

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

If a fatal error is detected and processing cannot be continued, the following message is output to the console, processing is stopped, and control is returned to the OS.

```
A>ST78K4 XXX.S  
Structured assembler preprocessor for RA78K/IV Vx.xx [xx xxx xx]  
Copyright (C) NEC Corporation 19xx,xxxx  
  
A006 File not found 'XXX.S'  
  
Program aborted.  
  
A>
```

## 8.4 Options

### 8.4.1 Types of options

Structured assembler options provide detailed instructions for structured assembler operations.

These structured assembler options are listed below.

**Table 8-2. Options List**

	Option	Specification format	Function
1	Device type specification	-C	Specifies the type of target device (chip)
2	Word symbol character specification	-SC	Specifies the last character in a word symbol name
3	Symbol definition	-D	Specifies a symbol such as a symbol assigned to an #IFDEF directive
4	Tab number specification	-WT	Specifies the position for outputting a converted instruction
5	Include file specification	-I	Specifies the drive and directory for an include file
6	Secondary source file specification	-O	Specifies the name of a secondary source file
7	Error list file specification	-E	Specifies the name of an error list file
8	Parameter file specification	-F	Specifies the name of a parameter file
9	Symbol name length specification	-S -NS	Specifies the length of (number of valid characters in) a symbol
10	Debug information output specification	-GS -NGS	Specifies output of structured assembler source-level debug information
11	Secondary source file forced output specification	-J	Specifies forced output of a secondary source file
12	Kanji code specification	-ZS -ZE -ZN	Specifies the type of kanji code used in comment statements
13	Device file search path specification	-Y	Specifies the path to use when searching for device files
14	Help specification	--	Outputs the contents of st78k4.hlp to the console

### 8.4.2 Option specification method

**(1) Option name**

Option names are not case-sensitive (can be entered in upper-case or lower-case letters).

Do not enter blank spaces after the option mark in an option name.

**(2) Option specification position**

Options can be specified either before or after the input file specification.

**(3) Option parameters**

Parameters assigned to options should be specified after the option, with no blank spaces.

**(4) Duplicate specification of options**

When several specifications have been made for an option of the same name, the last specification takes priority (is valid).

### 8.4.3 Description of options

The following is a detailed description of the above options.

**(1) Device type specification (-C)****[Coding format]**

-C device type

Interpretation when omitted: cannot be omitted

**[Function]**

The -C option specifies the target device (processor chip) for the structured assembler.

**[Description]**

<1> Be sure to specify this option. The structured assembler executes preprocessing for a specified target device and generates source code for the assembler.

An error occurs if the -C option is omitted.

<2> A warning occurs if the -C option and the processor model specification control instruction specify different processor models. In such cases, this pre-processor gives priority to the model specified by the option.

<3> The model specified by the -C option is output to a secondary source file as a processor model specification control instruction. However, it is not output if the same model has been specified by a processor model specification control instruction.

**[Use example]**

```
A>st78k4 test.s -c4026
```

**[Caution]**

The -C option cannot be omitted. However, specification at the command line can be omitted if a processor model specification control instruction (\$PROCESSOR) exists at the start of the source file.

For details of processor model specification control instructions, see “**CHAPTER 6 CONTROL INSTRUCTIONS**”.

Also, see “**Cautions on the use of device files**” with regard to device types.

**(2) Word symbol character specification (-SC)****[Coding format]**

-SC character

Interpretation when omitted: P or p

**[Function]**

Specifies the last character in the symbol to be judged when byte/word separation is required in symbol names.

**[Description]**

<1> The structured assembler generates different instructions depending on whether the data is being handled in bytes or words. For example, for an assignment, a MOV instruction is created for byte data and a MOVW instruction is created for word data.

If the data includes a reserved word for a word symbol, (such as `rp` or `sfrp` for `AX`; see “**2.3 Reserved Words**”), a word instruction is generated.

<2> If the specified symbol does not include a reserved word, the last character in the symbol is judged to determine whether the symbol is a word symbol or a byte symbol, and the corresponding instruction is then generated.

<3> If the -SC option is not specified, any symbol ending in “P” or “p” is judged as a word symbol.

<4> Only English alphabet letters can serve as characters to be judged in this case. There is no distinction between upper-case and lower-case letters.

<5> If specified more than once, the most recent specification takes priority.

**[Use examples]**

A>st78k4 test.s -sc@

**<TEST.S>**

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

**<TEST.ASM>**

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



**(3) Symbol definition specification (-D)****[Coding format]**

-D symbol name [= numerical value] [,symbol name [= numerical value] ...]

**[Function]**

This option defines a symbol.

**[Description]**

- <1> The numerical values assigned to symbols can be expressed as binary, octal, decimal, or hexadecimal numbers.  
If the numerical value specification is omitted, a value of "1" is taken.
- <2> This option has the same result as defining a symbol via the #define directive.
- <3> Up to 30 symbols can be specified via the command line if separated by commas.
- <4> This option is usually used in conjunction with the #ifdef directive.
- <5> If this option is specified more than once, the most recent specification takes priority.
- <6> If both this option and a #define directive are specified, a warning message is output and only the definition made via the #define directive is valid.
- <7> Specifications made using English alphabet letters are not case-sensitive.

**[Use example]**

A>st78k4 test.s -dTRUE=1

**(4) Tab number specification (-WT)****[Coding format]**

- WT numerical value 1
- WT [numerical value 1], numerical value 2
- WT [numerical value 1], [numerical value 2], numerical value 3

**[Function]**

This option specifies the number of tabs until the output position of the converted assembly language.

**[Description]**

- <1> The -WT option can be used to freely change the instruction output position for the assembler source to improve program readability.
- <2> Numerical value 1 specifies the number of tabs (default = 2) until the output position of the instruction. Numerical value 2 specifies the number of tabs (default = 3) until the output position of the instruction's operands. Numerical value 3 specifies the number of tabs (default = 4) until the output position of the instruction's comments.
- <3> Specify all numerical values as decimal values within the following ranges.
  - Numerical value 1: 0 to 97
  - Numerical value 2: 1 to 98
  - Numerical value 3: 2 to 99
  - Numerical value 1 < numerical value 2 < numerical value 3
- <4> If this option is specified more than once, the most recent specification takes priority.

**[Use example]**

A>st78k4 test.s -wt3,4,5

**(5) Include file specification (-I)****[Coding format]**

-I [drive name:] directory

**[Function]**

This option specifies an include file to be input to the structured assembler.

**[Description]**

- <1> This option specifies the drive number and directory where the target include file is located.
- <2> If the -I option is omitted, the include file is assumed to be in the current drive and current directory.
- <3> If this option is specified more than once, the most recent specification takes priority.

**[Use examples]**

A>st78k4 test.s -ib:\include

TEST.S : Located in the same directory as the structured assembler

FILE : Located in drive B's "INCLUDE" directory.

**<TEST.ST>**

```
#include    "FILE"

          A=SIZE1
          B=SIZE2
```

**<FILE>**

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

**<TEST.ASM>**

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

**(6) Secondary source file specification (-O)****[Coding format]**

-O [ [drive name:] directory] file name]

**[Function]**

This option specifies the output destination for a converted secondary source file.

**[Description]**

- <1> This option specifies the drive name, directory, and file name for the converted secondary source file.
- <2> If the -O option is omitted, the output file is created in the current directory, using the input file but replacing the file type with “.ASM”.
- <3> “NUL” and “AUX” can be used as file names.
- <4> The secondary source file is not output if a fatal error has occurred.
- <5> If this option is specified more than once, the most recent specification takes priority.

**[Use example]**

A>st78k4 test.s -osample.asm

**(7) Error list file specification (E)****[Coding format]**

-E [ [ (drive name:) directory] file name]

**[Function]**

This option specifies the output destination for an error list file.

**[Description]**

- <1> This option specifies the drive name, directory, and file name where the error list file will be output.
- <2> If the -E option is omitted, the output file is created in the current directory, using the input file but replacing the file type with ".EST".
- <3> "NUL" and "AUX" can be used as file names.
- <4> If this option is specified more than once, the most recent specification takes priority.

**[Use example]**

A>st78k4 test.s -esample.est

**(8) Parameter file specification (-F)****[Coding format]**

-F [ [drive name:] directory] file name

**[Function]**

This option specifies the file name of a parameter file.

**[Description]**

- <1> This option specifies the drive name, directory, and file name for the output parameter file.
- <2> The file name cannot be omitted. If the file type is omitted, it is interpreted as “.PST”.
- <3> When combined with the -D option, several symbols can be defined on the command line.
- <4> An error occurs if this option is specified more than once.
- <5> Parameter file nesting is prohibited. An error occurs if it is attempted.
- <6> Any character that appears after a “,” or “#” and before the LH or EOF is interpreted as a comment.

**[Use examples]**

<TEST.S>

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

<SAMPLE.PST>

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

A>st78k4 -fsample.pst

As a result, TEST.S is a processing target for the structured assembler. The secondary source file is generated as TEST.ASM and the error list file is generated as SAMPLE.EST.

**(9) Symbol name length specification (-S/-NS)****[Coding format]**

-S  
-NS

**[Function]**

This option specifies the maximum symbol name length.

**[Description]**

- <1> The -S option specifies a maximum symbol name length of 31 characters.
- <2> The -NS option sets eight characters as the length of a symbol name whose default maximum symbol name length is 31 characters.
- <3> If the -S/-NS options are omitted, the -S option specification is taken.
- <4> Valid symbols include symbols defined via a symbol definition directive, symbols accessed by a conditional directive, and user symbols.
- <5> If the -S and -NS options have both been specified, the second option to be specified takes priority.
- <6> If the -S option has been specified and \$NOSYMLEN has been entered in the input source file, \$SYMLEN is output to the secondary source file.
- <7> If the -NS option has been specified and \$SYMLEN has been entered in the input source file, \$NOSYMLEN is output to the secondary source file.

**[Use example]**

A>st78k4 test.s -ns

**(10) Debug information output specification (-GS/-NGS)****[Coding format]**

-GS  
-NGS

**[Function]**

This option specifies output of structured assembler source-level debug information.

**[Description]**

- <1> The -GS option specifies output of debug information to a secondary source file.
- <2> The -NGS option renders the -GS option invalid.
- <3> When the -GS option has been specified, "\$" is replaced by "," at the start of any compiler debug information that is in the input source file.
- <4> When the -GS and -NGS options have both been specified, the most recent option to be specified takes priority.
- <5> If this option omitted, specification of the -GS option is assumed.

**[Caution]**

When debugging at the structured assembler source level, use the structured assembler to specify the debug information output specification option (-G). When assembling a secondary source file, do not use the debug information output specification option (-G/-GA). The structured assembler outputs the necessary options as control instructions in secondary source files.

**[Use example]**

A>st78k4 test.s -gs



**(11) Secondary source file forced output specification (-J)****[Coding format]**

-J

**[Function]**

This option specifies forced output of a secondary source file when execution has ended due to a fatal error.

**[Description]**

- <1> This option specifies forced output of a secondary source file when execution has ended due to a fatal error.
- <2> The line containing the fatal error is output to a secondary source file as an input file image (i.e., as is).

**[Use example]**

A>st78k4 test.s -j

**(12) Kanji code specification (-ZS/-ZE/-ZN)****[Coding format]**

-ZS  
-ZE  
-ZN

**[Function]**

This option specifies the type of kanji code entered in comments.

**[Description]**

<1> Kanji codes are specified as shown below.

**Table 8-3. Interpretation of Kanji Code Specifications**

Option	Interpretation
-ZS	Interpreted as Shift-JIS code
-ZE	Interpreted as EUC code
-ZN	Not interpreted as a kanji code

<2> These options correspond to the following kanji code specification control instructions.

**Table 8-4. Corresponding Control Instructions**

Kanji code specification option	Kanji code specification control instruction
-ZS	\$KANJI CODE SJIS
-ZE	\$KANJI CODE EUC
-ZN	\$KANJI CODE NONE

<3> For details of priorities among kanji code specifications, see “**1.3.2 Environment variables**”.

**[Use example]**

A>st78k4 test.s -zs

**(13) Device file search path specification (-Y)****[Coding format]**

- Y Drive name
- Y [drive name] directory

**[Function]**

This option specifies the search path for device files.

**[Description]**

- <1> The device file is read via the specified path.
- <2> An error occurs if the specified path name is not valid.
- <3> Even if the directory specification symbol (Note 1) is not entered at the end of the directory specification, it is interpreted as having been entered.
- <4> Device file search paths are selected in the following order.
  - (a) Path specified by -Y option
  - (b) \.\dev (path via which st78k4.exe was launched)
  - (c) Path via which st78k4.exe was launched
  - (d) Current path
  - (e) Path specified by the "PATH" environment variable.

**Note 1.** The "¥" symbol used by PC-9800 Series computers is replaced by the "\" (back slash) symbol in IBM PCs.

**[Use example]**

A>st78k4 test.s -ya:\nectools\dev

**(14) Help specification (--)****[Coding format]**

--

**[Function]**

This option outputs help messages.

**[Description]**

- <1> The contents of st78k4.hlp are output to the console.
- <2> All other structured assembler options are rendered invalid.

**[Use example]**

A>st78k4--

The contents of st78k4.hlp are shown below.

Usage : st78k4 [option[...]] input-file [option[...]]

The option is as follows ([ ] means omissible, ... means repetition).

```
-cx           :Select target chip. (x = 4021,4026,etc.) *Must be specified.
-o[file]      :Create the assembler source file [with the specified name].
-e[file]      :Create the error list file [with the specified name].
-ffile        :Input options or source file name from specified file.
-idirectory   :Set include search path.
-s/-ns        :Expand symbol length up to 31 / or symbol length is 8.
-sc[character]:Specify the last character of word symbol.
-wtn1/-wt[n1],n2/-wt[n1],[n2],n3
               :Specify the number of tabs up to output position of each field.
               n1:Output position mnemonic field.
               n2:Output position operand field.      *Must be
               n3:Output position comment field.      0 <= n1 < n2 < n3 < 100.
-dname[=data][,name[=data][...]]
               :Define name [with data].
-gs/-ngs      :Output the structured assembler source debug information to
               assembler source file / Not.
-j            :Create the assembler source file if fatal error occurred.
-zs/-ze/-zn    :Change source regulation.
               -zs:SJIS code usable in comment.
               -ze:EUC code usable in comment.
               -zn:no multibyte code in comment.
-ydirectory   :Set device file search path.
--            :Show this message.
DEFAULT ASSIGNMENT:      -o -e -s -scp -wt2,3,4 -gs
```

## CHAPTER 9 INPUT/OUTPUT FILES

The structured assembler uses two types of input/output files.

- Input source program files
- Include files

There are also two types of output files.

- Secondary source program files
- Error list files

### 9.1 Input Source Program Files

The structured assembler's input files are files coded in assembly language that exist among the structured assembly language files to be processed.

The coding format for input source programs is shown in the following example of an input source program file.

#### (Example)

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

## 9.2 Include Files

Contents of a file that is not an input source program file can be brought without revision into an input source program file. Such files are called “include files”. Include files can contain highly versatile symbol declarations which enable them to be read into various source programs.

The contents of the selected include file are read into the source program file at the position where an #include directive has been entered.

The following examples show an input source program that contains an #include directive and an include file program.

### (Examples)

#### <Input source program>

```
#include "FILE"
```

```
A=SIZE1
```

```
B=SIZE2
```

#### <Include file>

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

```
#define SIZE2 #0AH
```

#### <Output source program>

```
MOV A,#08H ; A=#08H
```

```
MOV B,#0AH ; B=#0AH
```

### 9.3 Secondary Source Program Files

Secondary source program files are source program files that are output by the structured assembler to become input source program files for the assembler.

The following table lists ways in which secondary source program files are developed by the structured assembler.

**Table 9-1. Development by Structured Assembler**

Input source program file	Secondary source program file
Structured assembler control statement Structured assembler expression statement	Output as comments
Structured assembler directive	Not output
#INCLUDE	Contents of include file are output
Source defined as alias via #IFDEF	Not output
Comments	Output as comments
Other lines	Output without revision

The following shows an example of secondary source program file development.

**<Input source program>**

```
$processor (4026)
#include #FILE"
    if (A==#0)
        TMOD0=#0Ch
    endif
    CALL !XXX
    A=SIZE1
    B=SIZE2
    END
```

**<Include file>**

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

**<Secondary source program file>**

```
$processor(4026)
$NODEBUGA
$KANJICODE SJIS
$TOL_INF 2FH,110H,0,0FFFFH
$DGS FIL_NAM, .file,    U, 0FFFEH, 2FH, 67H, 1, 0
$DGS AUX_FIL, TTT.S
$DGL 0,2
$DGL 0,3
        CMP        A,#0        ;        if (A==#0)
        BNZ        $?L1
$DGL 0,4
        MOV        TMOD0,#0Ch        ;        TMOD0=#0Ch
$DGL 0,5
?L1:                ;        endif
$DGL 0,6
        CALL        !XXX        ;        CALL        !XXX
$DGL 0,7
        MOV        A,#08H        ;        A=#08H
$DGL 0,8
        MOV        B,#0AH        ;        B=#0AH
$DGL 0,9
        END                ;        END

; Target chip : uPD784026
; Device file : Vx.xx
```



## 9.4 Error List Files

Error list files contain error messages that are output when the structured assembler is launched.

The following shows an example of an error list file.

### <Input source program>

```
$processor (4026)
    if (A==#0)
        TMOD0=#0Ch
    END
```

### <Launch method>

```
A>st78k4 sample.s
```

```
Structured assembler preprocessor for RA78K/IV Vx.xx [ xx xxx xx]
  Copyright (C) NEC Corporation 19xx,xxxx
```

```
start
```

```
TTT.S(4) : F221 Missing ENDIF
```

```
Target chip : uPD784026
```

```
Device file : Vx.xx
```

```
Conversion complete, 1 error(s) found.
```

### <Error list file>

```
TTT.S(4) : F221 Missing ENDIF
```

```
Conversion complete, 1 error (s) found.
```

[MEMO]

## CHAPTER 10 ERROR MESSAGES

### 10.1 Overview of Error Messages

The ST78K4's error messages are divided into the following three levels.

#### <1> Abort error

Error messages at this level are output when execution fails during startup or when execution halts while in progress. As soon as the error message has been output, the program's execution is stopped, an error status is returned, and control is returned to the OS.

#### <2> Fatal error

Error messages at this level are output when a specification error has been made in the source program. If the error is discovered, the secondary source file is deleted. However, if the -J option has been specified, the source lines are output without revision to a secondary source file. In contrast to abort errors, processing continues after output of a fatal error message.

#### <3> Warning

Warning messages are output to advise the user against current processing which is not recommended. There is no error per se, and the current processing is allowed to continue.

The output format of the above levels of error messages is described below.

#### <1> Error that occurs during startup line

Error number □ error message

#### <2> Error that occurs any other time

Input file name (nnnn) □ : □ error number □ error message

The input file name can be an input source file name or an include file name. "nnnn" indicates the input file's line number.

## 10.2 Abort Errors

Table 10-1. Abort Error Messages (1/3)

A001	Message	Missing input file
	Cause	Input file was not specified
	User's response	Specify the input file
A002	Message	Too many input files
	Cause	Two or more input files were specified
	User's response	Specify only one input file
A004	Message	Illegal file specification for "file name"
	Cause	An illegal file was specified
	User's response	Specify the correct file name.
A005	Message	Illegal file specification for "file name"
	Cause	An illegal file was specified
	User's response	Specify the correct file name.
A006	Message	File not found for "file name"
	Cause	Specified file name does not exist.
	User's response	Specify an existing file name.
A008	Message	File specification conflicted for "file name"
	Cause	The same input/output file name was specified twice.
	User's response	Specify a unique file name for the input/output file.
A009	Message	Unable to make file "file name"
	Cause	The specified file has been write-protected.
	User's response	Remove the file's write protection.
A010	Message	Directory not found for "file name"
	Cause	A nonexistent drive or directory was specified in the output file name.
	User's response	Specify an existing drive and directory.
A011	Message	Illegal path for "option"
	Cause	The option for specifying the path name parameter was used to specify something other than a path name.
	User's response	Specify the correct path.
A012	Message	Missing parameter for "option"
	Cause	Required parameter was not specified
	User's response	Specify the parameter.
A014	Message	Out of range for "option"
	Cause	The specified numerical value was out of range.
	User's response	Specify a numerical value that is within the range.
A015	Message	Parameter is too long for "option"
	Cause	The character string used to specify the option parameter exceeds the maximum number of characters.
	User's response	Specify an option whose character string does not exceed the maximum.
A016	Message	Illegal parameter for "option"
	Cause	Syntax error in option parameter
	User's response	Specify the correct parameter.

Table 10-1. Abort Error Messages (2/3)

A017	Message	Too many parameters for “option”
	Cause	The total number of option parameters exceeds the maximum number allowed.
	User's response	Specify no more than the maximum number of parameters.
A018	Message	Option is not recognized for “option”
	Cause	Error in option specification
	User's response	Specify the correct option name.
A019	Message	Parameter file nested
	Cause	The -F option was specified within a parameter file.
	User's response	Do not specify the -F option in a parameter file.
A020	Message	Parameter file read error for “file name”
	Cause	Unable to read parameter file
	User's response	Specify the correct parameter file.
A021	Message	Memory allocation failed
	Cause	Not enough available memory
	User's response	Make more memory available
A101	Message	Open/read/write/close error on “file name”
	Cause	Error occurred during input or output of file, unable to open, read, write, or close the file.
	User's response	Specify the correct file name.
A102	Message	Can't find 'file name'
	Cause	Either the include file does not exist or the same name has been used for the include file and the input or output file.
	User's response	Specify the correct path, directory, and file name.
A103	Message	Illegal include file “file name”
	Cause	An illegal file name was specified for an include file.
	User's response	Specify the correct file name.
A104	Message	Illegal (-sc) character
	Cause	A character that cannot be used as a symbol was specified for the -SC option.
	User's response	Specify a legal character.
A105	Message	Can't define the reserved symbol
	Cause	A reserved word symbol was specified for the -D option.
	User's response	Do not specify a reserved word symbol for the -D option.
A106	Message	Duplicate PROCESSOR control
	Cause	The PROCESSOR control instruction was specified twice in the source file. The specified model is different from the model specified by the -C option.
	User's response	Specify the PROCESSOR control instruction only once. Revise the model name specification.
A107	Message	No processor specified
	Cause	No device (processor chip) type was specified.
	User's response	Specify the device type.
A108	Message	Illegal processor type specified
	Cause	An illegal device type was specified in a PROCESSOR control instruction in the source file.
	User's response	Specify a legal device type.

**Table 10-1. Abort Error Messages (3/3)**

A109	Message	Illegal processor type specified -C
	Cause	An illegal device type was specified via the -C option.
	User's response	Specify a legal device type.
A110	Message	Can't use this control outside module header
	Cause	A control instruction that should be entered in the source module header has been entered in an ordinary source line.
	User's response	Enter control instructions in the source module header.
A111	Message	Syntax error in module header
	Cause	The control instruction entered in the source module header contains a syntax error.
	User's response	Enter the control instruction using the correct syntax (coding format).
A112	Message	Structured assembler preprocessor internal error
	Cause	An error has occurred within the structured assembler itself.
	User's response	Contact your local NEC representative.

## 10.3 Fatal Errors

Table 10-2. Fatal Error Messages (1/3)

F201	Message	Illegal #ELSE/#ENDIF
	Cause	#ELSE and #ENDIF statements have been entered in illegal positions.
	User's response	Specify #ELSE and #ENDIF statements in legal positions.
F202	Message	Illegal #ENDCALLT
	Cause	An #ENDCALLT statement has been entered in an illegal position.
	User's response	Specify the #ENDCALLT statement in a legal position.
F203	Message	Missing #ENDIF
	Cause	#ENDIF statement does not exist
	User's response	Enter an #ENDIF statement in the correct position.
F204	Message	Missing #ENDCALLT
	Cause	#ENDCALLT statement does not exist
	User's response	Enter an #ENDCALLT statement in the correct position
F205	Message	Too many #DEFCALLT definition
	Cause	The limit for registering callt instruction conversion patterns has been exceeded.
	User's response	Reduce the number registered via #defcallt.
F206	Message	Too many CALL instructions
	Cause	Too many instructions have been defined via #DEFCALLT and #ENDCALLT.
	User's response	Define only one instruction via #DEFCALLT and #ENDCALLT.
F207	Message	Duplicate definition
	Cause	The same conversion pattern has been defined twice.
	User's response	Revise the registration via #DEFCALLT.
F208	Message	Symbol table overflow
	Cause	The maximum number of symbols has been exceeded.
	User's response	Reduce the number of symbols.
F209	Message	Syntax error
	Cause	The entered statement contains a syntax error.
	User's response	Enter a correct statement.
F210	Message	Nest level error
	Cause	A nesting error has occurred. (Possible causes include an overflow or a nesting pair)
	User's response	Enter a correct control statement.
F211	Message	Too many characters in a line
	Cause	The maximum number of characters per line has been exceeded.
	User's response	Limit the number of characters per line to 218 or less.
F212	Message	Too many include files
	Cause	An include file contains an include directive.
	User's response	Do not specify an include directive in an include file.

Table 10-2. Fatal Error Messages (2/3)

F214	Message	Illegal BREAK
	Cause	A BREAK statement was entered in an illegal position.
	User's response	Enter the BREAK statement in a correct position.
F215	Message	Illegal CONTINUE
	Cause	A CONTINUE statement was entered in an illegal position.
	User's response	Enter the CONTINUE statement in a correct position.
F216	Message	Illegal CASE/DEFAULT/ENDS
	Cause	A CASE/DEFAULT/ENDS statement was entered in an illegal position.
	User's response	Enter the CASE/DEFAULT/ENDS statement in a correct position.
F217	Message	Illegal ELSEIF/ELSE/ENDIF
	Cause	An ELSE/ELSEIF/ENDIF statement was entered in an illegal position.
	User's response	Enter the ELSE/ELSEIF/ENDIF statement in a correct position.
F218	Message	Illegal NEXT
	Cause	A NEXT statement was entered in an illegal position.
	User's response	Enter the NEXT statement in a correct position.
F219	Message	Illegal ENDW
	Cause	An ENDW statement was entered in an illegal position.
	User's response	Enter the ENDW statement in a correct position.
F220	Message	Illegal UNTIL/UNTIL_BIT
	Cause	An UNTIL or UNTIL_BIT statement was entered in an illegal position.
	User's response	Enter the UNTIL or UNTIL_BIT statement in a correct position.
F221	Message	Missing ENDIF
	Cause	The ENDIF statement does not exist.
	User's response	Enter the ENDIF statement in a correct position.
F222	Message	Missing ENDS
	Cause	The ENDS statement does not exist.
	User's response	Enter the ENDS statement in a correct position.
F223	Message	Missing ENDW
	Cause	The ENDW statement does not exist.
	User's response	Enter the ENDW statement in a correct position.
F224	Message	Missing NEXT
	Cause	The NEXT statement does not exist.
	User's response	Enter the NEXT statement in a correct position.
F225	Message	Missing UNTIL/UNTIL_BIT
	Cause	The UNTIL or UNTIL_BIT statement does not exist.
	User's response	Enter the UNTIL or UNTIL_BIT statement in a correct position.
F226	Message	Illegal character in a line
	Cause	An illegal character was entered in a source line.
	User's response	Delete the illegal character that was entered in the source line.
F227	Message	Illegal operand in a line
	Cause	An incorrect data size was specified in an assignment statement or a comparison statement.
	User's response	Specify the correct data size.



**Table 10-2. Fatal Error Messages (3/3)**

F228	Message	Illegal SFR access in operand
	Cause	An access-disabled sfr symbol was specified in an assignment expression.
	User's response	Check the sfr symbol's access status and then enter a correct sfr symbol.
F229	Message	Symbol is reserved: "symbol name"
	Cause	A reserved symbol was used.
	User's response	Change the symbol name.

## 10.4 Warning Messages

**Table 10-3. Warning Messages**

W301	Message	Symbol redefinition
	Cause	A symbol was redefined via the #define statement.
	Program's response	The most recently defined symbol is valid.
	User's response	If you want to use the earlier symbol definition, revise the statement.
W302	Message	Duplicate PROCESSOR option and control
	Cause	Different device types were specified via the -C option and the \$PROCESSOR control instruction.
	Program's response	The device type specified via the -C option is valid and the device type specified via the \$PROCESSOR control instruction is ignored.
	User's response	Check whether or not the correct device type was specified via the -C option.

## CHAPTER 11 MAXIMUM PERFORMANCE

**Table 11-1. 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 <sup>(Note 1)</sup>
Logical operator operands	17 <sup>(Note 2)</sup>
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]

## 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	$\alpha == \beta$	True when $\alpha = \beta$ , false when $\alpha \neq \beta$	P.47
NotEqual	$\alpha != \beta$	True when $\alpha \neq \beta$ , false when $\alpha = \beta$	P.50
LessThan	$\alpha < \beta$	True when $\alpha < \beta$ , false when $\alpha \geq \beta$	P.53
GreaterThan	$\alpha > \beta$	True when $\alpha > \beta$ , false when $\alpha \leq \beta$	P.56
GreaterEqual	$\alpha \geq \beta$	True when $\alpha \geq \beta$ , false when $\alpha < \beta$	P.59
LessEqual	$\alpha \leq \beta$	True when $\alpha \leq \beta$ , 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)	$\alpha = \beta (\gamma)$	$(\gamma) \leftarrow \beta \quad \alpha \leftarrow (\gamma)$	
Sequential assign	$\alpha 1 = \dots = \alpha n = \beta$	$\alpha 1 \leftarrow \beta, \dots, \alpha n \leftarrow \beta$	
Sequential assign (with register specification)	$\alpha 1 = \dots = \alpha n = \beta (\gamma)$	$\gamma \leftarrow \beta, \alpha 1 \leftarrow \gamma, \dots, \alpha n \leftarrow \gamma$	
Increment assignment	$\alpha += \beta$	$\alpha \leftarrow \alpha + \beta$	P.87
Increment assignment (with register specification)	$\alpha += \beta$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma + \beta, \alpha \leftarrow \gamma$	
Increment assignment (with register specification)	$\alpha += \beta, CY$	$\alpha \leftarrow \alpha + \beta, CY$	
Increment assignment (with register specification)	$\alpha += \beta, CY$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma + \beta, CY, \alpha \leftarrow \gamma$	
Decrement assignment	$\alpha -= \beta$	$\alpha \leftarrow \alpha - \beta$	P.91
Decrement assignment (with register specification)	$\alpha -= \beta$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma - \beta, \alpha \leftarrow \gamma$	
Decrement assignment (with register specification)	$\alpha -= \beta, CY$	$\alpha \leftarrow \alpha - \beta, CY$	
Decrement assignment (with register specification)	$\alpha -= \beta, CY$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma - \beta, CY, \alpha \leftarrow \gamma$	
Multiple assign	$\alpha * = \beta$	$\alpha \leftarrow \alpha \times \beta$	P.95
Multiple assign	$\alpha * = \beta$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \times \beta, \alpha \leftarrow \gamma$	
Divide assign	$\alpha /= \beta$	$\alpha \leftarrow \alpha \div \beta$	P.98
Logical AND assignment	$\alpha \&= \beta$	$\alpha \leftarrow \alpha \cap \beta$	P.100

Table A-3. Expressions (2/2)

Expression	Coding format	Function	Page
Logical AND assignment (with register specification)	$\alpha \&= \beta$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \cap \beta, \alpha \leftarrow \gamma$	P.100
Logical OR assignment	$\alpha  = \beta$	$\alpha \leftarrow \alpha \cup \beta$	P.103
Logical OR assignment (with register specification)	$\alpha  = \beta$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \cup \beta, \alpha \leftarrow \gamma$	
Logical XOR assignment	$\alpha ^= \beta$	$\alpha \leftarrow \alpha \wedge \beta$	P.106
Logical XOR assignment (with register specification)	$\alpha ^= \beta$ (Register)	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma \wedge \beta, \alpha \leftarrow \gamma$	
Right shift (rotate) assignment	$\alpha >>= \beta$	( $\alpha$ shifted to right of $\beta$ bit)	P.109
Right shift assignment (with register specification)	$\alpha >>= \beta$ (Register)	$\gamma \leftarrow \alpha, (\gamma \text{ shifted to right of } \beta \text{ bit}), \alpha \leftarrow \gamma$	
Left shift assignment	$\alpha <<= \beta$	( $\alpha$ shifted to left of $\beta$ bit)	P.112
Left shift assignment (with register specification)	$\alpha <<= \beta$ (Register)	$\gamma \leftarrow \alpha, (\gamma \text{ shifted to left of } \beta \text{ bit}), \alpha \leftarrow \gamma$	
Increment	$\alpha ++$	$\alpha \leftarrow \alpha + 1$	P.115
Decrement	$\alpha --$	$\alpha \leftarrow \alpha - 1$	P.117
Exchange	$\alpha <->= \beta$	$\alpha \leftarrow \alpha <->= \beta$	P.119
Exchange (with register specification)	$\alpha <->= \beta$ ( $\gamma$ )	$\gamma \leftarrow \alpha, \gamma \leftarrow \gamma <-> \beta, \alpha \leftarrow \gamma$	
Set bit	$\alpha = 1$	$\alpha \leftarrow 1$	P.122
Set bit (with register specification)	$\alpha = 1$ (CY)	$CY \leftarrow 1, \alpha \leftarrow 1$	
Sequential set bit	$\alpha 1 = \dots = \alpha n = 1$	$\alpha n \leftarrow 1, \dots, \alpha 1 \leftarrow 1$	
Sequential set bit (with register specification)	$\alpha 1 = \dots = \alpha n = 1$ (CY)	$CY \leftarrow 1, \alpha n \leftarrow 1, \dots, \alpha 1 \leftarrow 1$	
Clear bit	$\alpha = 0$	$\alpha \leftarrow 0$	P.125
Clear bit (with register specification)	$\alpha = 0$ (CY)	$CY \leftarrow 0, \alpha \leftarrow 0$	
Sequential clear bit	$\alpha 1 = \dots = \alpha n = 0$	$\alpha n \leftarrow 0, \dots, \alpha 1 \leftarrow 0$	
Sequential clear bit (with register specification)	$\alpha 1 = \dots = \alpha n = 0$ (CY)	$CY \leftarrow 0, \alpha n \leftarrow 0, \dots, \alpha 1 \leftarrow 0$	

Table A-4. Directives

Directive	Coding format	Page
#define	#define symbol character string	P.130
#ifdef	#ifdef symbol text 1 #else text 2 #endif	P.132
#include	#include "file name"	P.134
#defcallt	#defcallt CALLT table label CALL label #endcallt	P.136

[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) $\alpha, \beta$ BNZ        \$?LFALSE	lower case letters	P.47
	CMP(W) $\alpha, \beta$ BZ         \$?LTRUE BR         ?LFALSE ?LTRUE:	upper case letters	
$\alpha == \beta (\gamma)$	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BNZ        \$?LFALSE	lower case letters	
	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BZ         \$?LTRUE BR         LFALSE ?LTRUE:	upper case letters	
$\alpha \neq \beta$	CMP(W) $\alpha, \beta$ BZ         \$?LFALSE	lower case letters	P.50
	CMP(W) $\alpha, \beta$ BNZ        \$?LTRUE BR         ?LFALSE ?LTRUE:	upper case letters	
$\alpha \neq \beta (\gamma)$	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BZ         \$?LFALSE	lower case letters	
	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BNZ        \$?LTRUE BR         ?LFALSE ?LTRUE:	upper case letters	
$\alpha < \beta$	CMP(W) $\alpha, \beta$ BNC        \$?LFALSE	lower case letters	P.53
	CMP(W) $\alpha, \beta$ BC         \$?LTRUE BR         ?LFALSE ?LTRUE:	upper case letters	
$\alpha < \beta (\gamma)$	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BNC        \$?LFALSE	lower case letters	
	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ 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) $\alpha, \beta$ BNH \$?LFALSE	lower case letters	P.56
	CMP(W) $\alpha, \beta$ BH \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha > \beta (\gamma)$	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BNH \$?LFALSE	lower case letters	
	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BH \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha > = \beta$	CMP(W) $\alpha, \beta$ BC \$?LFALSE	lower case letters	P.59
	CMP(W) $\alpha, \beta$ BNC \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha > = \beta (\gamma)$	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BC \$?LFALSE	lower case letters	
	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ 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
$\alpha \leq \beta$	CMP(W) $\alpha, \beta$ BH         \$?LFALSE	lower case letters	P.62
	CMP(W) $\alpha, \beta$ BNH         \$?LTRUE BR           ?LFALSE ?LTRUE:	upper case letters	
$\alpha \leq \beta (\gamma)$	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BH         \$?LFALSE	lower case letters	
	MOV(W) $\gamma, \alpha$ CMP(W) $\gamma, \beta$ BNH         \$?LTRUE BR           ?LFALSE ?LTRUE:	upper case letters	

$\gamma$ : 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      \$?LTRUE BR      ?LFALSE ?LTRUE:	upper case letters (CY)	
	BZ      \$?LTRUE BR      ?LFALSE ?LTRUE:	upper case letters (Z)	
	BT      bit symbol, \$?LTRUE BR      ?LFALSE ?LTRUE:	upper case letters	
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      \$?LTRUE BR      ?LFALSE ?LTRUE:	upper case letters (CY)	
	BNZ      \$?LTRUE BR      ?LFALSE ?LTRUE:	upper case letters (Z)	
	BF      bit symbol, \$?LTRUE BR      ?LFALSE ?LTRUE:	upper case letters	

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

Logic expression	Generated instruction	Control statement condition	Page
$\alpha == \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNZ \$?LFALSE	lower case letters	P.75
	CMP(W) $\alpha, \beta$ BZ \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha != \beta \ \&\&$	CMP(W) $\alpha, \beta$ BZ \$?LFALSE	lower case letters	
	CMP(W) $\alpha, \beta$ BNZ \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha < \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNC \$?LFALSE	lower case letters	
	CMP(W) $\alpha, \beta$ BC \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha > \beta \ \&\&$	CMP(W) $\alpha, \beta$ BNH \$?LFALSE	lower case letters	
	CMP(W) $\alpha, \beta$ BH \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha >= \beta \ \&\&$	CMP(W) $\alpha, \beta$ BC \$?LFALSE	lower case letters	
	CMP(W) $\alpha, \beta$ BNC \$?LTRUE BR ?LFALSE ?LTRUE:	upper case letters	
$\alpha <= \beta \ \&\&$	CMP(W) $\alpha, \beta$ BH \$?LFALSE	lower case letters	
	CMP(W) $\alpha, \beta$ BNH \$?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
	BZ        \$?LTRUE BR        ?LFALSE ?LTRUE:	upper case letters	
bit symbol &&	BF        bit symbol, \$?LFALSE	lower case letters	
	BT        bit symbol, \$?LTRUE BR        ?LFALSE ?LTRUE:	upper case letters	
!CY &&	BC        \$?LFALSE	lower case letters	
	BNC      \$?LTRUE BR        ?LFALSE ?LTRUE:	upper case letters	
!Z &&	BZ        \$?LFALSE	lower case letters	
	BNZ      \$?LTRUE BR        ?LFALSE ?LTRUE:	upper case letters	
!bit symbol &&	BT        bit symbol, \$?LFALSE	lower case letters	
	BF        bit symbol, \$?LTRUE BR        ?LFALSE ?LTRUE:	upper case letters	
$\alpha == \beta$	CMP(W) $\alpha, \beta$ BZ        \$?LFALSE	lower case letters	P.78
$\alpha != \beta$	CMP(W) $\alpha, \beta$ BNZ      \$?LFALSE	upper case letters	
$\alpha < \beta$	CMP(W) $\alpha, \beta$ BC        \$?LFALSE	lower case letters	
$\alpha > \beta$	CMP(W) $\alpha, \beta$ BH        \$?LFALSE	upper case letters	
$\alpha >= \beta$	CMP(W) $\alpha, \beta$ BNC      \$?LFALSE	lower case letters	
$\alpha <= \beta$	CMP(W) $\alpha, \beta$ BNH      \$?LFALSE	upper case letters	
CY	BC        \$?LFALSE	lower case letters	
Z	BZ        \$?LFALSE	upper case letters	
bit symbol	BT        bit symbol, \$?LFALSE	lower case letters	
!CY	BNC      \$?LFALSE	upper case letters	
!Z	BNZ      \$?LFALSE	lower case letters	
!bit symbol	BF        bit symbol, \$?LFALSE	upper case letters	

Table B-4. Expressions (1/4)

Expression	Generated instruction	Page
$\alpha = \beta$	MOV $\alpha\ 1, \beta$	P.83
	MOVW $\alpha\ 1, \beta$	
	MOVG $\alpha\ 1, \beta$	
	MOV1 $\alpha\ 1, \beta$	
$\alpha = \beta\ (\gamma)$	MOV $\gamma, \beta$	
	MOV $\alpha\ 1, \gamma$	
	MOVW $\gamma, \beta$	
	MOVW $\alpha\ 1, \gamma$	
	MOVG $\gamma, \beta$	
	MOVG $\alpha\ 1, \gamma$	
	MOV1 $\gamma, \beta$	
	MOV1 $\alpha\ 1, \gamma$	
$\alpha += \beta$	ADD $\alpha, \beta$	P.87
	ADDW $\alpha, \beta$	
	ADDG $\alpha, \beta$	
	ADDWG $\alpha, \beta$	
$\alpha += \beta\ (\gamma)$	MOV $\gamma, \alpha$	
	ADD $\gamma, \beta$	
	MOV $\alpha, \gamma$	
	MOVW $\gamma, \alpha$	
	ADDW $\gamma, \beta$	
	MOVW $\alpha, \gamma$	
	MOVG $\gamma, \alpha$	
	ADDG $\gamma, \beta$	
	MOVG $\alpha, \gamma$	
$\alpha += \beta, CY$	ADDC $\alpha, \beta$	
$\alpha += \beta, CY\ (\gamma)$	MOV $\gamma, \alpha$	
	ADDC $\gamma, \beta$	
	MOV $\alpha, \gamma$	

Table B-4. Expressions (2/4)

Expression	Generated instruction	Page
$\alpha -= \beta$	SUB $\alpha, \beta$	P.91
	SUBW $\alpha, \beta$	
	SUBG $\alpha, \beta$	
	SUBWG $\alpha, \beta$	
$\alpha -= \beta (\gamma)$	MOV $\gamma, \alpha$	
	SUB $\gamma, \beta$	
	MOV $\alpha, \gamma$	
	MOVW $\gamma, \alpha$	
	SUBW $\gamma, \beta$	
	MOVW $\alpha, \gamma$	
	MOVG $\gamma, \alpha$	
	SUBG $\gamma, \beta$	
	MOVG $\alpha, \gamma$	
$\alpha -= \beta, CY$	SUBC $\alpha, \beta$	
$\alpha -= \beta, CY (\gamma)$	MOV $\gamma, \alpha$	
	SUBC $\gamma, \beta$	
	MOV $\alpha, \gamma$	
$\alpha *= \beta$	MULU $\beta$	P.95
	MULUW $\beta$	
$\alpha *= \beta (\gamma)$	MOVW $AX, \alpha$	
	MULU $\beta$	
	MOVW $\alpha, AX$	
	MOVW $AX, \alpha$	
	MULUW $\beta$	
	MOVW $\alpha, AX$	
$\alpha /= \beta$	DIVUW $C$	P.98
	DIVUX $C$	
$\alpha \&= \beta$	AND1 $CY, \beta$	P.100
	AND $\alpha, \beta$	
$\alpha \&= \beta (\gamma)$	MOV1 $CY, \alpha$	
	AND1 $CY, \beta$	
	MOV1 $\alpha, CY$	
	MOV $\gamma, \alpha$	
	AND $\gamma, \beta$	
	MOV $\alpha, \gamma$	



Table B-4. Expressions (3/4)

Expression	Generated instruction		Page
$\alpha \mid= \beta$	OR1	CY, $\beta$	P.103
	OR	$\alpha, \beta$	
$\alpha \mid= \beta (\gamma)$	MOV1	CY, $\alpha$	P.106
	OR1	CY, $\beta$	
	MOV1	$\alpha$ , CY	
	MOV	$\gamma, \alpha$	
	OR	$\gamma, \beta$	
	MOV	$\alpha, \gamma$	
$\alpha \wedge= \beta$	XOR1	CY, $\beta$	P.106
	XOR	$\alpha, \beta$	
$\alpha \wedge= \beta (\gamma)$	MOV1	CY, $\alpha$	P.109
	XOR1	CY, $\beta$	
	MOV1	$\alpha$ , CY	
	MOV	$\gamma, \alpha$	
	XOR	$\gamma, \beta$	
	MOV	$\alpha, \gamma$	
$\alpha >>= \beta$	SHR	$\alpha, \beta$	P.109
	SHRW	$\alpha, \beta$	
$\alpha >>= \beta (\gamma)$	MOV	$\gamma, \alpha$	P.112
	SHR	$\gamma, \beta$	
	MOV	$\alpha, \gamma$	
	MOVW	$\gamma, \alpha$	
	SHRW	$\gamma, \beta$	
	MOVW	$\alpha, \gamma$	
$\alpha <<= \beta$	SHL	$\alpha, \beta$	P.112
	SHLW	$\alpha, \beta$	
$\alpha <<= \beta (\gamma)$	MOV	$\gamma, \alpha$	P.115
	SHL	$\gamma, \beta$	
	MOV	$\alpha, \gamma$	
	MOVW	$\gamma, \alpha$	
	SHLW	$\gamma, \beta$	
	MOVW	$\alpha, \gamma$	
$\alpha ++$	INC	$\alpha$	P.115
	INCW	$\alpha$	
	INCG	$\alpha$	
$\alpha --$	DEC	$\alpha$	P.117
	DECW	$\alpha$	
	DECG	$\alpha$	

Table B-4. Expressions (4/4)

Expression	Generated instruction		Page
$\alpha \leftrightarrow \beta$	XCH	$\alpha, \beta$	P.119
	XCHW	$\alpha, \beta$	
$\alpha \leftrightarrow \beta (\gamma)$	MOV	$\gamma, \alpha$	
	XCH	$\gamma, \beta$	
	MOV	$\alpha, \gamma$	
	MOVW	$\gamma, \alpha$	
	XCHW	$\gamma, \beta$	
	MOVW	$\alpha, \gamma$	
$\alpha = 1$	SET1	$\alpha 1$	P.122
$\alpha = 1$ (CY)	SET1	CY	
	SET1	$\alpha 1$	
$\alpha = 0$	CLR1	$\alpha 1$	P.125
$\alpha = 0$ (CY)	CLR1	CY	
	CLR1	$\alpha 1$	

## 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

### Hong Kong, Philippines, Oceania

NEC Electronics Hong Kong Ltd.  
Fax: +852-2886-9022/9044

### Asian Nations except Philippines

NEC Electronics Singapore Pte. Ltd.  
Fax: +65-250-3583

### Europe

NEC Electronics (Europe) GmbH  
Technical Documentation Dept.  
Fax: +49-211-6503-274

### Korea

NEC Electronics Hong Kong Ltd.  
Seoul Branch  
Fax: 02-528-4411

### Japan

NEC Semiconductor Technical Hotline  
Fax: 044-435-9608

### South America

NEC do Brasil S.A.  
Fax: +55-11-6462-6829

### Taiwan

NEC Electronics Taiwan Ltd.  
Fax: 02-2719-5951

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	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Technical Accuracy	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>