User's Manual

RA78K0 Ver. 3.80

Assembler Package

Structured Assembly Language

Target Devices
78K0 Series

Document No. U17197EJ1V0UM00 (1st edition)
Date Published October 2004 CP(K)

[MEMO]

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC Electronics product in your application, please contact the NEC Electronics 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.

[GLOBAL SUPPORT]
http://www.necel.com/en/support/support.html

NEC Electronics America, Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65030

- **Sucursal en España**
  Madrid, Spain
  Tel: 091-504 27 87
- **Succursale Française**
  Vélizy-Villacoublay, France
  Tel: 01-30-67 58 00
- **Filiale Italiana**
  Milano, Italy
  Tel: 02-66 75 41
- **Branch The Netherlands**
  Eindhoven, The Netherlands
  Tel: 040-244 58 45
- **Tyskland Filial**
  Taeby, Sweden
  Tel: 08-63 80 820
- **United Kingdom Branch**
  Milton Keynes, UK
  Tel: 01908-691-133

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-558-3737

NEC Electronics Shanghai Ltd.
Shanghai, P.R. China
Tel: 021-5888-5400

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

NEC Electronics Singapore Pte. Ltd.
Novena Square, Singapore
Tel: 6253-8311

J04.1

INTRODUCTION

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 **RA78K0 Assembler Package** (hereafter called **RA78K0**).

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 **RA78K0 Assembler Package User’s Manual** (**Language (U17198E)** and **Operation (U17199E)**).

Descriptions related to the RA78K0 in this manual apply to Ver. 3.80 or later.

[Target Readers]

This manual is intended for user engineers who understand the functions and instructions of the microcontroller (78K0 Series) subject to development.

Readers requiring a description of the functions of microcontrollers in the 78K0 Series should refer to the target chip’s User’s Manual.

[Organization]

This manual consists of the following six chapters and appendices:

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.

APPENDIXES.A SYNTAX LISTS
This appendix presents a structured assembler syntax list.

APPENDIXES.B LISTS OF GENERATED INSTRUCTIONS
This appendix presents a list of instructions generated by the structured assembler.

The instruction sets are not detailed in this manual. For these instructions, refer to the user’s manual of the microcontroller for which software is being developed.

Also, for instructions on architecture, refer to the user's manual (hardware version) of each microcontroller for which software is being developed.

[How to Read This Manual]

Those using an structured assembler for the first time are encouraged to read from **CHAPTER 1 GENERAL** of this manual. Those who have a general understanding of structured assembler may skip this chapter.
However, all readers should read section “1.3 Before Starting Program Development”.

[Conventions]

The following symbols and abbreviations are used throughout this manual:

:	Same format is repeated.
[ ]:	Characters enclosed in these brackets can be omitted.
{ }:	One of the items in { } is selected.
“ ”:	Characters enclosed in “ ”(quotation marks) are a character string.
‘ ’:	Characters enclosed in ‘ ’ (single quotation marks) are a character string.
( ):	Characters between parentheses are a character string.
< >:	Characters (mainly title) enclosed in these brackets are a character string.
__:	An underline is used to indicate an important point or input character strings.
Δ:	Indicates one or more blanks characters or tabs.
/:	Character delimiter
~:	Continuity
Boldface:	Characters in boldface are used to indicate an important point or reference point.

[Related Documents]

The documents (user's manuals) related to this manual are listed below.

The related documents indicated in this publication may include preliminary versions.

However, preliminary versions are not marked as such.

Document Name		Document No.
RA78K0 Ver. 3.80 Assembler Package	Operation	U17199E
	Language	U17198E
	Structured Assembly Language	This manual
CC78K0 Ver. 3.70 C Compiler	Operation	U17201E
	Language	U17200E
SM plus System Simulator	Operation	U17246E
	User Open Interface	U17247E
SM78K0 Series Ver. 2.52 System Simulator	Operation	U16768E
PM plus Ver. 5.20		U16934E
ID78K0-NS Ver. 2.52 Integrated Debugger	Operation	U16488E
ID78K0-QB Ver .2.81 Integrated Debugger	Operation	U16996E
78K0 Series	Instruction	U12326E

[MEMO]

CONTENTS

CHAPTER 1 GENERAL ... 13
- 1.1 Overview ... 13
- 1.2 Overview of Functions ... 14
  - 1.2.1 Main Functions ... 14
  - 1.2.2 Flowchart of Program Development ... 15
- 1.3 Before Starting Program Development ... 16
  - 1.3.1 Maximum performance ... 16
  - 1.3.2 Word symbols and byte symbols ... 17
  - 1.3.3 Definition of label ... 17

CHAPTER 2 SOURCE PROGRAM CODING METHODS ... 18
- 2.1 Basic Configuration of Source Programs ... 18
- 2.2 Source Program Elements ... 20
- 2.3 Reserved Words ... 23
- 2.4 Label Generation Rules ... 25
- 2.5 Size Specification ... 26
- 2.6 Data Sizes ... 27
- 2.7 Comments ... 28
- 2.8 Tool Information ... 29
- 2.9 Output Results of Input Source Files by ST78K0 ... 30

CHAPTER 3 CONTROL STATEMENTS ... 31
- 3.1 Control Statement Characters ... 31
- 3.2 Nesting ... 32
- 3.3 Register Specification ... 33
- 3.4 Control Statement Functions ... 35
  - 3.4.1 Conditional branch ... 36
  - 3.4.2 Conditional loop ... 46
- 3.5 Conditional Expressions ... 59
  - 3.5.1 Comparison expressions ... 60
  - 3.5.2 Test bit expressions ... 82
  - 3.5.3 Logical operations ... 89

CHAPTER 4 EXPRESSIONS ... 96
- 4.1 Overview of Expressions ... 96
- 4.2 Assignment Statements ... 99
- 4.3 Count Statements ... 124
- 4.4 Exchange Statements ... 128
- 4.5 Bit Manipulation Statements ... 130

CHAPTER 5 DIRECTIVES ... 136
- 5.1 Overview of Directives ... 136
- 5.2 Directive Functions ... 137
  - #DEFINE ... 138
  - #IFDEF/#ELSE/#ENDIF ... 140
  - #INCLUDE ... 142
  - #DEFCALLT ... 143

CHAPTER 6 CONTROL INSTRUCTIONS ... 144
- 6.1 Overview of Control Instructions ... 144
- 6.2 Assembler Control Instructions ... 145
- 6.3 Control Instruction Functions ... 147
  - $PROCESSOR ... 148
  - $KANJICODE ... 149

APPENDIX A SYNTAX LISTS ... 150

APPENDIX B LISTS OF GENERATED INSTRUCTIONS ... 155

APPENDIX C INDEX ... 165

LIST OF FIGURES

Figure No.   Title , Page

1-1   ST78K0 Function ... 14
1-2   Program Development Flowchart ... 15
3-1   Nesting example ... 32

LIST OF TABLES

Table No.	Title , Page
1-1	Maximum Performance of ST78K0 ... 16
2-1	Structured Assembly Language Coding ... 18
2-2	Alphanumeric Characters ... 20
2-3	Special Characters ... 21
2-4	Invalid Characters ... 22
2-5	Reserved Word ... 23
2-6	Data Sizes ... 27
2-7	Output by ST78K0 ... 30
3-1	List of Control statements ... 35
3-2	Generated Instructions for switch Statements ... 43
3-3	Generated Instructions for Comparison Instructions ... 60
3-4	Comparison Expressions ... 61
3-5	Test Bit Expressions ... 82
3-6	Logical Operations ... 89
3-7	Generated Instructions (Control Statement in Lower Case Letters) for Logical AND ... 90
3-8	Generated Instructions (Control Statement in Upper Case Letters) for Logical AND ... 91
3-9	Generated Instructions for Logical OR ... 94
4-1	Assignment Statements ... 96
4-2	Count Statements ... 98
4-3	Exchange Statements ... 98
4-4	Bit Manipulation Statements ... 98
4-5	Generated Instructions for Assignments ... 102
4-6	Generated Instructions for Increment Assignments ... 105
4-7	Generated Instructions for Decrement Assignments ... 108
4-8	Generated Instructions for Logical AND Assignments ... 114
4-9	Generated Instructions for Logical OR Assignments ... 116
4-10	Generated Instructions for Logical XOR Assignments ... 119
4-11	Generated Instructions for Increment ... 125
4-12	Generated Instructions for Decrement ... 127
4-13	Generated Instructions for Exchange ... 129
4-14	Generated Instructions for Set Bit ... 132
4-15	Generated Instructions for Clear Bit ... 135
5-1	List of Directives ... 137
6-1	Control Instructions that Can Be Entered Only in Module Headers ... 145
6-2	Control Instructions that Are Recognized as the Module Body ... 146
6-3	Control Instruction List ... 147
6-4	Interpretation of Kanji Code ... 149
A-1	Control Statements ... 150
A-2	Conditional Expressions ... 151
A-3	Expressions ... 152
A-4	Directives ... 153
A-5	Control Instructions ... 154
B-1	Generated Instructions for Comparison Expressions ... 155
B-2	Generated Instructions for Test Bit Expressions ... 158
B-3	Generated Instructions for Logic Expressions ... 159
B-4	Expressions ... 162

CHAPTER 1 GENERAL

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

1.1 Overview

The RA78K0 structured assembler preprocessor (ST78K0) is a program in the “RA78K0 Assembler Package” that is used for software development of microcontrollers in the 78K0 Series.

The ST78K0 converts structured assembly statements such as “if~else~endif” and “for~next” into assembly language source . Control statements are used to enter “if~else~endif” and “for~next” descriptions.

As such , the ST78K0 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.

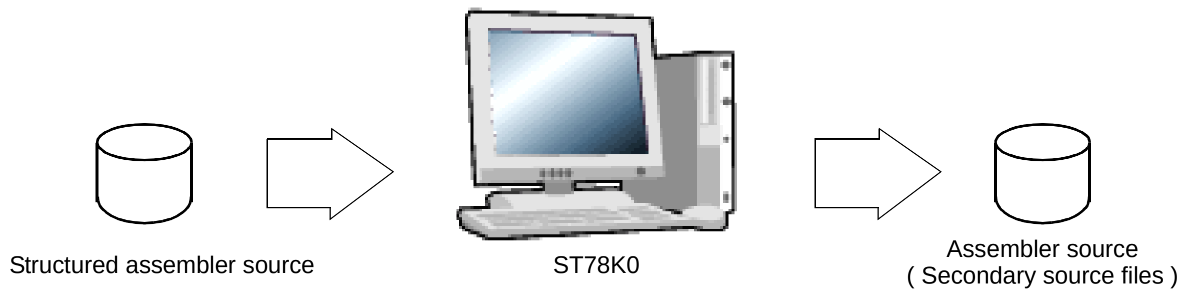
1.2 Overview of Functions

The ST78K0 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 that serves as an input source file for the assembler.

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.

Figure 1-1 ST78K0 Function



1.2.1 Main Functions

(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 ST78K0.

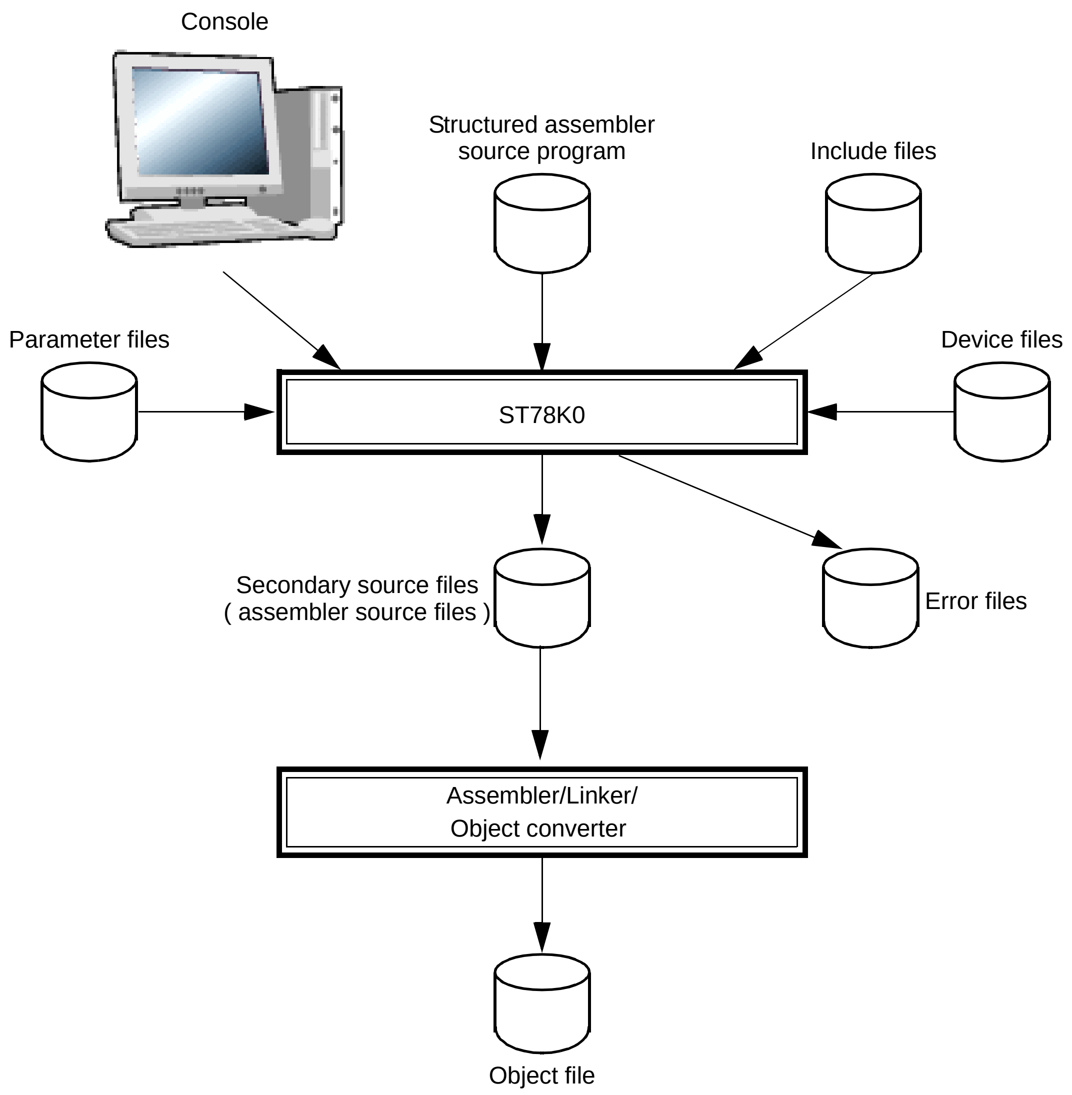
(6) A directive is provided for converting to CALLT instructions , so that routines can be registered to a CALLT table following development of a program.

(7) Easy-to-read assembly lists can be created by changing the assembler source output position.

1.2.2 Flowchart of Program Development

Figure 1-2 shows a flowchart of program development.

Figure 1-2 Program Development Flowchart


Console
Structured assembler source program
Include files
Parameter files
Device files
ST78K0
Secondary source files (assembler source files)
Error files
Assembler/Linker/ Object converter
Object file

Remark Obtain the device file by downloading it from the Online Delivery Service (ODS), which can be accessed from the following Website.

http://www.necel.com/micro/ods/eng/tool/DeviceFile/list.html

1.3 Before Starting Program Development

The maximum performance of the ST78K0 and points to be noted are described below.

1.3.1 Maximum performance

Table 1-1 Maximum Performance of ST78K0

Item	Maximum value
Line length (not including LF or CR)	2048 characters
Number of symbols registered in #define directive (excluding reserved words)	512 symbols
Character length of symbol registerd in #define directive	31 characters
Nesting levels in control statement	31 levels
Nesting levels in #ifdef directive	8 levels
#defcallt directives	32
Nesting of #include directives	Not supported
Number of redefinitions by #define directive	31 times
Number of operands assigned in a series	33 (Note 1)
Logical operator operands	17 (Note 2)
Number of symbols defined by option "-D"	30
Number of include file paths specifiable by -I option	64

Notes 1. The maximum value is expressed as follows.

S1 = S2 = ... S32 = S33

Up to 33 symbols and 32 equal (=) signs can be inserted.

Notes 2. The maximum value is expressed as follows.

expression 1&&expression 2&& ... &&expression 16&&expression 17

Up to 17 expressions and 16 “&&” (or “||”) signs can be inserted.

1.3.2 Word symbols and byte symbols

The ST78K0 uses the last character in each user symbol to determine whether the symbol is a word symbol or a byte symbol. The default character for word symbols is "-SCP" , and it can be changed via the -SC option.

For details of the -SC option , see the RA78K0 Assembler Package Operation User's Manual.

< Example 1 >

Structured assembler source

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

→

Assembler source

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

< Example 2 > Start command for ST78K0

```
C> ST78K0 INPUT.S -SC@
         |      |         |
         |      |        "@" is used as the character indicating a word symbol.
         |      |
         |     Input file specification
         |
     ST78K0 command name
```

Structured assembler source

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

→

Assembler source

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

1.3.3 Definition of label

When defining labels (symbol indicating address via assembler) , be sure to enter the label definition on a separate line from the ST78K0 statement.

< Example of incorrect coding >

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

< Example of correct coding >

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

CHAPTER 2 SOURCE PROGRAM CODING METHODS

This chapter describes coding methods for source programs etc.

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 RA78K0 Assembler Package Language User’s Manual.

Each line (between two LFs) can contain up to 2048 characters.

The types of coding used in structured assembly language are listed below in Table 2-1.

Table 2-1 Structured Assembly Language Coding

Type			Coding
ST78K0 statement	Control statement	Conditional branch	if ~ elseif ~ else ~ endif if_bit ~ elseif_bit ~ else ~ endif switch ~ case ~ default ~ ends

		Conditional loop	for ~ next
			while ~ endw
			while_bit ~ endw
			repeat ~ until
			repeat ~ until_bit

		Other	break , continue , goto
	Expression	Assignment statement	Assign (=) , assignment plus operation (+= , etc.) , shift (rotate) assignment (>>= , etc.)
		Count statement	Increment (++) , Decrement (--)
		Exchange statement	Exchange (<->)
Conditional expression		Comparison expression	== , != , < , > , >= , <=
		Test bit expression	Bit symbol , !bit symbol
		Logical operation	Logical AND (&&) , Logical OR (||)

(1) Control statements

Control statements include “if ~ elseif ~ else ~ endif”, "if_bit ~ elseif_bit ~ else ~ endif", and “switch ~ case ~ default ~ ends” statements that represent conditional branches , “for ~ next” , “while ~ endw”, "while_bit ~ endw", “repeat ~ until”, and "repeat ~ until_bit" 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”.

(3) Conditional expressions

Conditional expressions are entered as control statement conditions. For details , see “3.5 Conditional 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

Name		Character												
Numerals		0	1	2	3	4	5	6	7	8	9			
Letters	Upper case	A N	B O	C P	D Q	E R	F S	G T	H U	I V	J W	K X	L Y	M Z
	Lower case	a n	b o	c p	d q	e r	f s	g t	h u	i v	j w	k x	l y	m z

In the ST78K0 , only the first character in control statements are case-sensitive. Any lower case letters that appear after the first character are converted to upper case letters. However , secondary source files are output using the case specifications in which they were entered.

Table 2-3 Special Characters

Character	Name	Use
?	Question mark	Character used as letter
@	Unit price symbol	Character used as letter
_	Underlining	Character used as letter
	White space	Delimiter symbol for phrases
HT	Horizontal tab	Character used as white space
,	Comma	Delimiter symbol for operands
.	Period	Bit position symbol for bit symbols
“	Double quotation mark	Specification character for #INCLUDE directive's disk-type file names
‘	Single quotation mark	Symbol used to mark start and end of character constant
+	Plus symbol	Positive sign or increment operation
-	Minus symbol	Negative sign or decrement operation
*	Asterisk	Multiplication operation
/	Slash	Division operation
&	Ampersand	Logical AND operator
|	Separator symbol	Logical OR operator
ˆ	Upward arrow symbol	Exclusive OR operator
(	Left parenthesis	Change in operation sequence or expression in control statement
)	Right parenthesis	
=	Equal symbol	Assignment operator , comparison operator
:	Colon	Delimiter symbol for labels
;	Semicolon	Comment start symbol or delimiter symbol in control statement expressions
#	Sharp symbol	First character in ST78K0 directive or immediate display symbol
$	Dollar sign	Location or counter value Display symbol in control instruction
!	Exclamation point	Direct addressing specification symbol , negation display symbol
<	Not equal (less than) symbol	Comparison operator
>	Not equal (more than) symbol	
\	Back slash	Directory specification symbol
[	Left bracket	Indirect address specification symbol
]	Right bracket	
LF	Line feed	End of line symbol

An error will occur if any of the following Invalid Characters are entered.

Table 2-4 Invalid Characters

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

When an illegal character is entered , an error occurs and each illegal character is replaced by a period (.) when a secondary file is output.

However , invalid characters can be used in comments.

(2) Identifiers

Identifiers are names that are attached to numerical data , addresses , etc.

Identifiers are used to make the contents of source programs easier to identify.

Use #define statements to define details of identifiers (see also “5.2 Directive Functions”).

(3) Symbols

The last character in the symbol name determines whether the ST78K0 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 RA78K0 Assembler Package Language User’s Manual.

(5) Expressions

Expressions are constants , special characters , and symbols that are combined using operators (for details of assembly language expressions , see the RA78K0 Assembler Package Language User’s Manual.

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

< Examples >

- Coding method for assembler

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

- Coding method for ST78K0 structured assembler source program

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

2.3 Reserved Words

Table 2-5 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

Type	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 , BANKNUM
	DATAPOS , BITPOS , MASK

Table 2-5 Reserved Word

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

2.4 Label Generation Rules

When using control statements in assembler language instructions , the ST78K0 generates labels for branch instructions.

Labels generated by the ST78K0 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.

(1) Description format

(Δ size specification character Δ)

(2) Function

- If the size character is either “B” or “b” , the data size is changed to bytes.
- If the size character is “P” , “p” , “W” , or “w” , the data size is changed to words.

(3) Description

- An error will occur if the size specification character is incorrect.
- An error will occur if a size specification is entered in an assignment expression or a conditional expression which does not support size specifications.
- 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.
- When specifying a user symbol , be sure to change the data size to the specified data size.
- 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.
- Word access cannot be specified in size specifications.

2.6 Data Sizes

The ST78K0 checks the data size of symbols. This is because the symbols differ according to the instruction being generated. However , the ST78K0 allows the assembler to determine whether or not the symbol definitions and constants are entered correctly.

The data sizes checked by the ST78K0 are listed below.

Table 2-6 Data Sizes

Symbol in Generated Instruction Table	Description
a	CY
b	Bit symbols This ST78K0 recognizes bit sfrs and symbols entered using the format “α , β” as bit symbols. Items that can be entered as “α” include byte user symbols , word user symbols , byte-specified user symbols , sfrs , A , PSW , [HL] , and constants. Items that can be entered as “β” include byte user symbols , word user symbols , and constants.
c	Byte user symbols
d	Byte-specified user symbols , sfrs that overlaps saddr
e	A
f	Byte registers
g	sfr
h	PSW
i	Word user symbols
j	Word-specified user symbols
k	AX
l	BC , DE , HL
m	RP0 , RP1 , RP2 , RP3
n	sfrp
o	SP
p	Direct access specification symbols These are symbols that are specified using the format “!addr”. Byte user symbols , word user symbols , constants , and $ can be entered as “addr”.
q	Indirect access specification symbols These are symbols that are specified using the format “[HL]” , “[HL + byte]” , “[HL + B]” , and “[HL + C]”. Byte user symbols , constants , and $ can be entered as “byte”.
r	Special indirect access specification symbols These are symbols that are specified using the format “[DE]”.
s	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 ST78K0 outputs tool information.

If an input source file contains tool information that has been output by the ST78K0 , 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.

(1) Output format

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

2FH indicates that it is tool information output by the ST78K0 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 3.10 -> 310H

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

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

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

2.9 Output Results of Input Source Files by ST78K0

Input source files are output as follows by the ST78K0.

Table 2-7 Output by ST78K0

Input source program file	Secondary source program file
ST78K0 control statements ST78K0 expression statements	Output as comments
ST78K0 directives	Not output
#INCLUDE	Outputs include contents
Source alias set by #IFDEF	Not output
Comments	Output as comments
Other lines	Output as is

CHAPTER 3 CONTROL STATEMENTS

This chapter presents examples in describing control statement functions.

Control statements are used to structurally code the flow of program control (see also “3.4 Control Statement Functions”).

3.1 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 ST78K0 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.

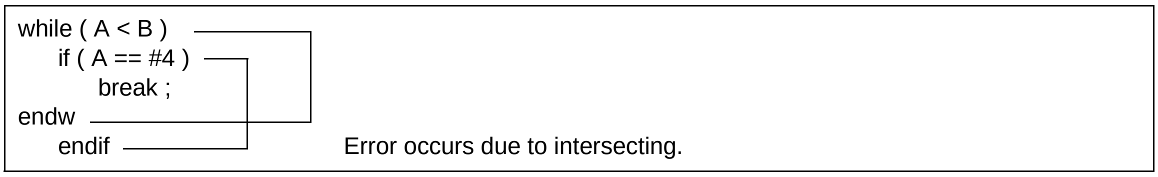
Paired control statements (such as “if , else , endif”) can have mixed upper case and lower case letters. In other words , it is possible to enter one as “IF ~ else ~ ENDIF”.

3.2 Nesting

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

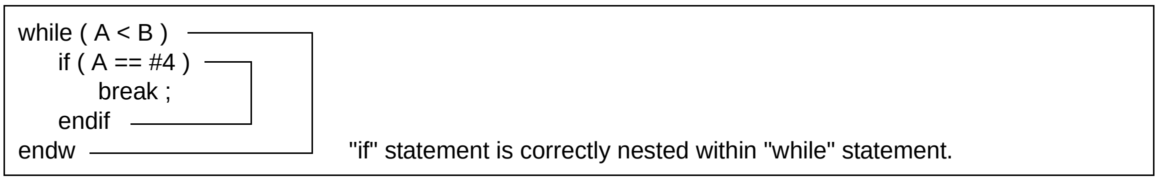
Figure 3-1 Nesting example

< Example of incorrect coding >



```
while ( A < B )
    if ( A == #4 )
        break ;
endw
    endif                     Error occurs due to intersecting.
```

< Example of correct coding >



```
while ( A < B )
    if ( A == #4 )
        break ;
    endif
endw                          "if" statement is correctly nested within "while" statement.
```

3.3 Register Specification

(1) Description format

([Δ] [=] [Δ] register name [Δ])

(2) Function

- 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 >

Output source	Input source

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

- 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 >

Output source	Input source

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

- 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 >

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

(3) Description

- 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.
- For a list of register names , see Table 2-5.

  sfr specifications can also be entered.
- The processing for an assignment statement within a for statement is the same as for comparison expressions.

3.4 Control Statement Functions

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

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

Table 3-1 List of Control statements

Type	Description	Remark
Conditional branch	if ~ elseif ~ else ~ endif	
	if_bit ~ elseif_bit ~ else ~ endif	
	switch ~ case ~ default ~ ends	
Conditional loop	for ~ next	(Repetition of increment specification)
	while ~ endw	(Repetition of conditional expression testing before processing)
	while_bit ~ endw	(Repetition of conditional expression testing before processing)
	repeat ~ until	(Repetition of conditional expression testing after processing)
	repeat ~ until_bit	(Repetition of conditional expression testing after processing)
	break	(Extraction of loop block)
	continue	(Repetition of loop block)
	goto	(Escape to go to exception processing)

3.4.1 Conditional branch

Conditional branch if

(1) if ~ elseif ~ else ~ endif

[Description 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]

- if ~ endif

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

  The if block may occupy several lines.

- 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.

- 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.

[Description]

- 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 "3.5 Conditional Expressions".

- if ~ else ~ endif is used when coding two branches for a condition.
- 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.
- elseif statements and else statements can be omitted and several elseif statements can be entered.

[Generated instructions]

(1) Processing of if (conditional expression)

- Generates an instruction to test the condition of the conditional expression.
- 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)

- Generates a branch instruction to an endif statement.
- Generates a label for the branch instruction generated by an if statement.
- Generates an instruction to test the condition of the conditional expression.
- Generates a branch instruction to branch to an elseif block or else block if the condition is not met.

(3) Processing of else

- Generates a branch instruction to an endif statement.
- Generates a label for the branch instruction generated by an if statement or elseif statement.

(4) Processing of endif

- Generates a label for the branch instruction generated by an if statement , elseif statement , or else statement.

(5) Additional description

- These blocks can be mixed with elseif_bit.

[Use examples]

(1) When entered in lower case letters

Output source	Input source
``` CMP A , #0```	if (A == #0)
``` BNZ $?L1```	
``` MOV1 CY , TFLG.0```	CY = TFLG.0
``` MOVW AX , #0FFH```	AX = #0FFH
``` BR ?L2```	
?L1 :	else
``` MOVW BC , #0A00H```	BC = #0A00H
?L2 :	endif

(2) When entered in upper case letters

Output source	Input source
``` CMP A , #0```	IF (A == #0)
``` BZ $?L3```	
``` BR ?L4```	
?L3 :	
``` MOV1 CY , TFLG.0```	CY = TFLG.0
``` MOVW AX , #0FFH```	AX = #0FFH
``` BR ?L5```	
?L4 :	ELSE
``` MOVW BC , #0A00H```	BC = #0A00H
?L5 :	ENDIF

Conditional branch if_bit

(2) if_bit ~ elseif_bit ~ else ~ endif

[Description format]

```
[ Δ ] if_bit [ Δ ] ( test bit expression 1 )
            if_bit block
[ Δ ] elseif_bit [ Δ ] ( test bit expression 2 )
            elseif_bit block
[ Δ ] else [ Δ ]
            else block
[ Δ ] endif [ Δ ]
```

[Function]

- if_bit ~ endif

  If conditional expression 1 is true , the if_bit block is executed.

  The if_bit block may occupy several lines.
- 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.
- 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.
- Additional description

  These blocks can be mixed with elseif.

[Description]

- Test bit expressions are entered as conditional expressions 1 and 2.

  For details of test bit expressions , see “3.5 Conditional Expressions”.
- 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.
- elseif_bit statements and else statements can be omitted and several elseif_bit statements can be entered.

[Generated instructions]

(1) Processing of if_bit (bit condition)

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

(2) Processing of elseif_bit (bit condition)

- Generates a branch instruction to an endif statement.
- Generates a label for the branch instruction generated by an if_bit statement.

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

(3) Processing of else

- Generates a branch instruction to an endif statement.
- Generates a label for the branch instruction generated by an if_bit statement or elseif_bit statement.

(4) Processing of endif

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

[Use examples]

(1) When entered in lower case letters

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

(2) When entered in upper case letters

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

Conditional branch switch

(3) switch ~ case ~ default ~ ends

[Description 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]

- If the value of the case symbol matches the case constant , the specified statement is executed.
- 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.
- Normally , a break statement must be entered to skip a switch block.

[Description]

- The possible specifications for “case symbol” depend on the assembly language of the target device.
- If a break statement is not entered , a comparison instruction is executed for the next case statement.
- Constants can be expressed as binary , octal , decimal , hexadecimal , or character string constants. However , since the ST78K0 recognizes constants as character strings , be careful to use only constants that the assembler can recognize as such.
- The case symbol is transferred to the specified register only when a register specification has been made.

[Generated instructions]

(1) Processing of switch statement

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

(b) If a register has been specified , the case symbol is transferred to the specified register.
However , an error occurs if a comparison instruction cannot be generated.
For details , see Table 3-2.

(2) Processing of case statement

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

(b) CMP or CMPW is generated , and if the specified constant does not match , a branch instruction for another case statement , default statement , or ends statement is generated.

?LTRUE :	Branch destination label when specified constant matches
?LFALSE :	Branch destination label when specified constant does not match

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

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

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

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

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

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

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

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

(3) Processing of default statement

Generates a label for the branch instruction from the case statement

(4) Processing of ends statement

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

Table 3-2 Generated Instructions for switch Statements

Case symbol		Without register specification	With register specification										
			a	b	e	f	g	h	k	l	m	n	o
a	CY												
b	Bit symbol												
c	Byte user symbol	*3			*1				*2				
d	Byte data	*3			*1								
e	A	*3											
f	Byte register	*1			*1								
g	sfr	*1			*1								
h	PSW	*1			*1								
i	Word user symbol	*2			*1				*2				
j	Word data	*2							*2				
k	AX	*3											
l	BC , DE , HL	*2							*2				
m	RP0 , RP1 , RP2 , RP3												
n	sfrp	*2							*2				
o	SP	*2							*2				
p	Direct access symbol	*1			*1				*2				
q	Indirect access symbol	*1			*1								
r	[DE]	*1			*1								
s	Immediate symbol	*1			*1				*2				

*1 : Generates MOV instruction

*2 : Generates MOVW instruction

*3 : Does not generate transfer instruction

Empty columns indicate errors.

[Use examples]

(1) When entered in lower case letters

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

(2) When entered in upper case letters

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

3.4.2 Conditional loop

Conditional loop for

(4) for ~ next

[Description 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.

[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 (b).

(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 (c).

(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.

- Use saddr instead of a register name as the control variable in expression 1 and expression 3.
- When specifying a register , specify either A or AX.
- 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.

[Use examples]

(1) When entered in lower case letters

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

(2) When entered in upper case letters

Output source	Input source
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

[Description format]

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

[Function]

- The instruction group is repeatedly executed as long as the conditional expression remains true.

[Description]

- 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.
- As the register name , specify the register used in the comparison expression or logic expression entered as “(conditional expression)”.
- 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
- Generates a label for the branch instruction generated by endw.
- Generates a condition testing instruction. If a register name has been specified , the specified register is used when generating the condition testing instruction.
- Generates a branch instruction for removing the while (conditional expression) statement from the while block when the condition tests as false.

(2) endw
- Generates a branch instruction for an execution loop.
- Generates a label for the branch instruction that is used to remove endw from the while block.

[Use examples]

(1) When entered in lower case letters

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

(2) When entered in upper case letters

Output source	Input source
?L3 :	WHILE (AX < #0FFFH)
CMPW AX , #0FFFH	
BC $?L4	
?L4 :	
MOV B , #0FH	B = #0FH
NCW HL	HL++
BR ?L3	
?L5 :	ENDW

Conditional loop while_bit

(6) while_bit ~ endw

[Description format]

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

[Function]

- The instruction group can be executed as long as the bit condition is true.

[Description]

- 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
  - Generates a label for the branch instruction generated by endw.
  - Generates an instruction for testing the bit condition as true or false.
  - 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
  - Generates a branch instruction for an execution loop.
  - Generates a label for the branch instruction that is used to remove endw from the while_bit block.

[Use examples]

(1) When entered in lower case letters

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

(2) When entered in upper case letters

Output source	Input source
?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

[Description format]

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

[Function]

- The instruction group is repeatedly executed as long as the conditional expression remains true.

[Description]

- 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.
- As the register name , specify the register used in the comparison expression or logic expression entered as “(conditional expression)”.
- 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
  - Generates a label for the branch instruction generated by until.

(2) Processing of until (conditional expression) statement
  - Generates a condition testing instruction for the conditional expression.
  - 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.

[Use examples]

(1) When entered in lower case letters

Output source	Input source

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

	                  CALL    !XXX
	         endif
	         CNT++
	until ( CNT == #0FFH )
	```

(2) When entered in upper case letters

Output source	Input source
?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) repeat ~ until_bit

[Description format]

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

[Function]

- The instruction group is repeatedly executed as long as the bit condition is false.

[Description]

- 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

- Generates a label for the branch instruction generated by until_bit.

(2) Processing of until_bit (bit condition)

- 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

Output source	Input source

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

(2) When entered in upper case letters

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

Conditional loop break

(9) break

[Description format]

```
[ Δ ] break
```

[Function]

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

[Description]

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

[Generated instructions]

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

```
BR        ?Lxxxx
```

[Use example]

Output source	Input source
?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

[Description 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]

- This is used to skip subsequent processing from the middle of a block and execute the next loop.
- 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]

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

Conditional loop goto

(11) goto

[Description format]

```
[ Δ ] goto Δ label
```

[Function]

- Unconditionally branches to a label.

[Description]

- 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.
- 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 ST78K0. Note also that the ST78K0 does not automatically check whether or not a branch destination label exists.

[Use examples]

Output source	Input source

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

3.5 Conditional Expressions

Conditional expressions are used to set conditions via control statements.

The following are examples of conditional expressions.

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

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

γ specifies the register that is used for this comparison.

3.5.1 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 “3.3 Register Specification” for a description of the register specification Description format.

The ST78K0 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 following pages describe the functions of the various comparison expressions.

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

Table 3-3 Generated Instructions for Comparison Instructions

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY																			
	b	Bit symbol																			
	c	Byte user symbol																			*1
	d	Byte data																			*1
	e	A		*1	*1	*1	*1			*1								*1	*1		*1
	f	Byte register				*1															
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX																			*2
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

*1 : Generates CMP instruction

*2 : Generates CMPW instruction

Empty columns indicate errors.

Table 3-4 Comparison Expressions

Comparison Expression	Description Format	Function
Equal (==)	$\alpha == \beta$	True when $\alpha = \beta$, false when $\alpha \neq \beta$
NotEqual (!=)	$\alpha != \beta$	True when $\alpha \neq \beta$, false when $\alpha = \beta$
LessThan (<)	$\alpha < \beta$	True when $\alpha < \beta$, false when $\alpha >= \beta$
GreaterThan (>)	$\alpha > \beta$	True when $\alpha > \beta$, false when $\alpha <= \beta$
GreaterEqual (>=)	$\alpha >= \beta$	True when $\alpha >= \beta$, false when $\alpha < \beta$
LessEqual (<=)	$\alpha <= \beta$	True when $\alpha <= \beta$, false when $\alpha > \beta$
FOREVER (forever)	forever	Endlessly loops the loop statement

Comparison expressions Equal (==)

(1) Equal (==)

[Description format]

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

[Function]

- When there is no register specification
  True when the contents of α and β are equal , false when they are not equal.
- When there is a register specification
  The contents of α are transferred to the specified register. True is the result when the contents of the specified register are equal to the contents of β and false is the result when they are not equal.

[Description]

- When there is no register specification
  For α and β , be sure to specify contents that can be entered in CMP or CMPW.
- When there is a register specification
  For α , be sure to specify contents that can be entered in MOV or MOVW.
  For β , be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

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

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

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

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

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

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

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

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

For details of combinations of α and β , see Table 3-3. α indicates the specified register. For further description of generated instructions for MOV , see “4.2 (1) Assign (=)”.

[Use examples]

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

Output source	Input source
CMPW AX , #0F0FH	if (AX == #0F0FH)
BNZ $?L1	
CALL !XXX	CALL !XX
BR ?L2	
?L1 :	else
CALL !YYY	CALL !YYY
?L2 :	endif

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

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

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

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

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

Output source	Input source
`        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 (!=)

[Description format]

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

[Function]

- When there is no register specification
  True when the contents of α and β are not equal , false when they are equal.
- When there is a register specification
  The contents of α are transferred to the specified register. True is the result when the contents of the specified register are not equal to the contents of β and false is the result when they are equal.

[Description]

- When there is no register specification
  For α and β , be sure to specify contents that can be entered in CMP or CMPW.
- When there is a register specification
  For α , specify contents that can be entered in MOV or MOVW.
  For β , specify contents that can be entered in CMP or CMPW.

[Generated instructions]

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

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

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

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

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

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

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

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

For details of combinations of α and β , see Table 3-3. α indicates the specified register. For further description of generated instructions for MOV , see “4.2 (1) Assign (=)”.

[Use examples]

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

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

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

Output source	Input source
MOV A , !XYZ	if (!XYZ != #5 (A))
CMP A , #5	
BZ $?L	
CALL !PPP	CALL !PPP
?L3 :	endif

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

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

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

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

Comparison expressions LessThan (<)

(3) LessThan (<)

[Description format]

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

[Function]

- When there is no register specification
  True when the contents of α are less than the contents of β , false when otherwise (i.e. , equal to or greater than).
- When there is a register specification
  The contents of α are transferred to the specified register. True is the result when the contents of the specified register are less than the contents of β and false is the result when they are otherwise.

[Description]

- When there is no register specification
  For α and β , be sure to specify contents that can be entered in CMP or CMPW.
- When there is a register specification
  For α , be sure to specify contents that can be entered in MOV or MOVW.
  For β , be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

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

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

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

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

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

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

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

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

For details of combinations of α and β , see Table 3-3. α indicates the specified register. For further description of generated instructions for MOV , see “4.2 (1) Assign (=)”.

[Use examples]

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

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

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

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

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

Output source	Input source
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

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

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

Comparison expressions GreaterThan (>)

(4) GreaterThan (>)

[Description format]

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

[Function]

- When there is no register specification
  True when the contents of α are greater than the contents of β , false when otherwise (i.e. equal to or less than).
- When there is a register specification
  The contents of α are transferred to the specified register. True is the result when the contents of the specified register are greater than the contents of β and false is the result when they are otherwise.

[Description]

- When there is no register specification
  For α and β , be sure to specify contents that can be entered in CMP or CMPW.
- When there is a register specification
  For α , be sure to specify contents that can be entered in MOV or MOVW.
  For β , be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

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

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

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

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

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

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

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

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

For details of combinations of α and β , see Table 3-3. α indicates the specified register. For further description of generated instructions for MOV , see "4.2 (1) Assign (=)".

[Use examples]

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

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

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

Output source	Input source
MOVW AX , ABCP	if (ABCP > #0FE40H (AX))
CMPW AX , #0FE40H	
BZ $?L3	
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

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

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

Output source	Input source
MOVW AX , ABCP	IF (ABCP > #0FE40H (AX))
CMPW AX , #0FE40H	
BZ $$ + 4	
BNC $?L7	
R ?L8	
?L7 :	
CALL !PPP	CALL !PPP
?L8 :	ENDIF

Comparison expressions GreaterEqual (>=)

(5) GreaterEqual (>=)

[Description format]

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

[Function]

- When there is no register specification
  True when the contents of α are greater than or equal to the contents of β , false when they are less than the contents of β.
- When there is a register specification
  The contents of α are transferred to the specified register. True is the result when the contents of the specified register are greater than or equal to the contents of β and false is the result when they are less than the contents of β.

[Description]

- When there is no register specification
  For α and β , be sure to specify contents that can be entered in CMP or CMPW.
- When there is a register specification
  For α , be sure to specify contents that can be entered in MOV or MOVW.
  For β , be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

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

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

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

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

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

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

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

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

For details of combinations of α and β , see Table 3-3. α indicates the specified register. For further description of generated instructions for MOV , see "4.2 (1) Assign (=)".

[Use examples]

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

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

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

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

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

Output source	Input source
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

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

Output source	Input source

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




	        CALL   !PPP
	ENDIF
	```

Comparison expressions LessEqual (<=)

(6) LessEqual (<=)

[Description format]

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

[Function]

- When there is no register specification
  True when the contents of α are less than or equal to the contents of β , false when they are greater than the contents of β.
- When there is a register specification
  The contents of α are transferred to the specified register. True is the result when the contents of the specified register are less than or equal to the contents of β and false is the result when they are greater than the contents of β.

[Description]

- When there is no register specification
  For α and β , be sure to specify contents that can be entered in CMP or CMPW.
- When there is a register specification
  For α , be sure to specify contents that can be entered in MOV or MOVW.
  For β , be sure to specify contents that can be entered in CMP or CMPW.

[Generated instructions]

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

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

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

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

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

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

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

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

For details of combinations of α and β , see Table 3-3. α indicates the specified register. For further description of generated instructions for MOV , see “4.2 (1) Assign (=)”.

[Use examples]

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

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

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

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

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

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

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

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

Comparison expressions FOREVER (forever)

(7) FOREVER (forever)

[Description 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

Output source	Input source

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

	        A = i
	        CALL    !XXX
	        if ( AX == #0FFH )

	                break
	        endif


	next
	```

(2) while statement

Output source	Input source

```
?L4 :
        MOV     A , i
        CALL    !XXX
        CMPW    AX , #0ffH
        BNZ     $?L5
        BR      ?L6
?L5 :
        BR      ?L4
?L6 :
```
	```
	while ( forever )
	        A = i
	        CALL    !XXX
	        if ( AX == #0ffH )

	                break
	        endif

	endw
	```

(3) repeat statement

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

3.5.2 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 ST78K0 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 ST78K0 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 following pages describe the functions of the various test bit expressions.

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

Table 3-5 Test Bit Expressions

Test Bit Expression	Description Format	Function
Bit symbol	Bit symbol	True when specified bit is 1
!bit symbol	!bit symbol	True when specified bit is 0

Test bit expressions Positive logic (bit)

(1) Bit symbol

[Description format]

```
[ Δ ] bit symbol [ Δ ]
```

[Function]

- True when the bit symbol contents are 1 , false when they are 0.
- The following control statements are able to include bit symbols entered as conditional expressions.

```
if        if_bit
elseif    elseif_bit
while     while_bit
until     until_bit
for
```

[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 :
```

(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

Output source	Input source
BNC $?L1	if_bit (CY)
CALL !XXX	CALL !XXX
BR ?L2	
?L1 :	else
CALL !YYY	CALL !YYY
?L2 :	endif
BNZ $?L3	if_bit (Z)
CALL !XXX	CALL !XXX
BR ?L4	
?L3 :	else
CALL !YYY	CALL !YYY
?L4 :	endif
BF TRFG.0 , $?L5	if_bit (TRFG.0)
CALL !XXX	CALL !XXX
BR ?L6	
?L5 :	else
CALL !YYY	CALL !YYY
?L6 :	endif

(2) When the control statement is entered in upper case letters

Output source	Input source
BC $?L7	IF_BIT (CY)
BR ?L8	
?L7 :	
CALL !XXX	CALL !XXX
BR ?L9	ELSE
?L8 :	CALL !YYY
CALL !YYY	ENDIF
?L9 :	
BZ $?L10	IF_BIT (Z)
BR ?L11	
?L10 :	
CALL !XXX	CALL !XXX
BR ?L12	
?L11 :	ELSE
CALL !YYY	CALL !YYY
?L12 :	ENDIF
BT TRFG.0 , $?L13	IF_BIT (TRFG.0)
BR ?L14	
?L13 :	
CALL !XXX	CALL !XXX
BR ?L15	
?L14 :	ELSE
CALL !YYY	CALL !YYY
?L15 :	ENDIF

Test bit expressions Negative logic (bit)

(2) !bit symbol

[Description format]

```
[ Δ ] !bit symbol [ Δ ]
```

[Function]

- True when the bit symbol contents are 0 , false when they are 1.
- The following control statements are able to include bit symbols entered as conditional expressions.

```
if        if_bit
elseif    elseif_bit
while     while_bit
until     until_bit
for
```

[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 :
```

(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

Output source	Input source
BC $?L1	if_bit (!CY)
CALL !XXX	CALL !XXX
BR ?L2	
?L1 :	else
CALL !YYY	CALL !YYY
?L2 :	endif
BZ $?L3	if_bit (!Z)
CALL !XXX	CALL !XXX
BR ?L4	
?L3 :	else
CALL !YYY	CALL !YYY
?L4 :	endif
BT TRFG.0 , $?L5	if_bit (!TRFG.0)
CALL !XXX	CALL !XXX
BR ?L6	
?L5 :	else
CALL !YYY	CALL !YYY
?L6 :	endif

(2) When the control statement is entered in upper case letters

Output source	Input source
BNC $?L7	IF_BIT (!CY)
BR ?L8	
?L7 :	
CALL !XXX	CALL !XXX
BR ?L9	ELSE
?L8 :	CALL !YYY
CALL !YYY	ENDIF
?L9 :	
	IF_BIT (!Z)
BNZ $?L10	
BR ?L11	
?L10 :	
CALL !XXX	CALL !XXX
BR ?L12	
?L11 :	ELSE
CALL !YYY	CALL !YYY
?L12 :	ENDIF
BF TRFG.0 , $?L13	IF_BIT (!TRFG.0)
BR ?L14	
?L13 :	
CALL !XXX	CALL !XXX
BR ?L15	
?L14 :	ELSE
CALL !YYY	CALL !YYY
?L15 :	ENDIF

3.5.3 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 ST78K0 generates branch instructions beginning from the highest-priority logical operator.

(1) Code example

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

The following pages describe the functions of the various logical operations.

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

Table 3-6 Logical Operations

Logical Operation	Describe Format	Function
Logical AND (&&)	Conditional expression 1 && conditional expression 2	True if both conditional expression 1 and conditional expression 2 are true
Logical OR (||)	Conditional expression 1 || conditional expression 2	True if either conditional expression 1 or conditional expression 2 is true

Logical operations Logical AND (&&)

(1) Logical AND (&&)

[Description format]

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

[Function]

- The logical AND result of conditional expression 1 and conditional expression 2 is obtained. The result is true when conditional expression 1 and conditional expression 2 are both true and the result is false otherwise. The entered operation is performed when two conditions are met.
  The output instruction differs depending on whether the control statement is entered in lower case letters or upper case letters.
  Instructions for testing are generated first for contents enclosed in parentheses "( )".

[Generated instructions]

(1) When the control statement is entered in lower case letters

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

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

(2) When the control statement is entered in upper case letters

Table 3-8 Generated Instructions (Control Statement in Upper Case Letters) for Logical AND

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

Conditional expression	Generated instruction

!Z &&	`BNZ $?LTRUE`
	`BR ?LFALSE`
	`?LTRUE :`

[Use examples]

(1) When the control statement is entered in lower case letters

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

(2) When the control statement is entered in upper case letters

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

Logical operations Logical OR (||)

(2) Logical OR (||)

[Description 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 3-9 Generated Instructions for Logical OR

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

[Use examples]

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

CHAPTER 4 EXPRESSIONS

This chapter describes the functions of the expressions.

4.1 Overview of 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

The functions of these expressions are described below.

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

Table 4-1 Assignment Statements

Assignment statement		Description format	Function
Assign (=)			
	Assign	$\alpha = \beta$	α <- β
	Sequential assign	$\alpha_1 = ... = \alpha_n = \beta$	α_1 = <- β , ... , α_n <- β
	Assign (with register specification)	$\alpha = \beta (\gamma)$	(γ) <- β , α <- (γ)
	Sequential assign (with register specification)	$\alpha_1 = ... = \alpha_n = \beta (\gamma)$	γ <- β , α_1 <- γ , ... , α_n <- γ
IncrementAssign (+=)			
	Increment assignment	α += β	α <- $\alpha + \beta$
	Increment assignment (with register specification)	α += β (register)	γ <- α , γ <- $\gamma + \beta$, α <- γ
	Increment assignment with carry	α += β , CY	α <- $\alpha + \beta$, CY
	Increment assignment with carry (with register specification)	α += β , CY (register)	γ <- α , γ <- $\gamma + \beta$, CY , α <- γ

Table 4-1 Assignment Statements

Assignment statement	Description format	Function
DecrementAssign (-=)		
Decrement assignment	α -= β	α <- α - β
Decrement assignment (with register specification)	α -= β , (register)	γ <- α , γ <- γ - β , α <- γ
Decrement assignment with carry	α -= β , CY	α <- α - β , CY
Decrement assignment with carry (with register specification)	α -= β , CY (register)	γ <- α , γ <- γ - β , CY , α <- γ
MultiplicationAssign (*=)		
Multiplication assignment	α *= β	α <- α * β
Multiplication assignment (with register specification)	α *= β (register)	γ <- α , γ <- γ * β , α <- γ
DivisionAssign (/=)		
Division assignment	α /= β	α <- α / β
Division assignment (with register specification)	α /= β (register)	γ <- α , γ <- γ / β , α <- γ
LogicalANDAssign (&=)		
Logical AND assignment	α &= β	α <- α ∩ β
Logical AND assignment (with register specification)	α &= β (register)	γ <- α , γ <- γ ∩ β , α <- γ
LogicalORAssign (|=)		
Logical OR assignment	α |= β	α <- α U β
Logical OR assignment (with register specification)	α |= β (register)	γ <- α , γ <- γ U β , α <- γ
LogicalXORAssign (^=)		
Logical XOR assignment	α ^= β	α <- α ^ β
Logical XOR assignment (with register specification)	α ^= β (register)	γ <- α , γ <- γ ^ β , α <- γ
RightShiftAssign (>>=)		
Right shift (rotate) assignment	α >>= β	(α shifted to right of β bit)
Right shift assignment (with register specification)	α >>= β (register)	γ <- α , (γ shifted to right of β bit) , α <- γ
LeftShiftAssign (<<=)		
Left shift assignment	α <<= β	(α shifted to left of β bit)
Left shift assignment (with register specification)	α <<= β (register)	γ <- α , (γ shifted to left of β bit) , α <- γ

Table 4-2 Count Statements

Count statement	Description format	Function
Increment (++)	α ++	α <- $\alpha + 1$
Decrement (--)	α --	α <- $\alpha - 1$

Table 4-3 Exchange Statements

Exchange statement		Description format	Function
Exchange (<->)			
	Exchange	α <-> β	α <- α <-> β
	Exchange (with register specification)	α <-> β (γ)	γ <- α , γ <- γ <-> β , α <- γ

Table 4-4 Bit Manipulation Statements

Bit manipulation statement		Description format	Function
Set bit (=)			
	Set bit	$\alpha = 1$	α <- 1
	Sequential set bit	α_1 = ... = α_n = 1	α_n = <- 1 , ... , α_1 <- 1
	Set bit (with register specification)	α = 1 (CY)	CY <- 1 , α <- 1
	Sequential set bit (with register specification)	α_1 = ... α_n = 1 (CY)	CY <- 1 , α_n <- 1 , ... , α_1 <- 1
Clear bit (=)			
	Clear bit	$\alpha = 0$	α <- 0
	Sequential clear bit	α_1 = ... = α_n = 0	α_n <- 0 , ... , α_1 <- 0
	Clear bit (with register specification)	α = 0 (CY)	CY <- 0 , α <- 0
	Sequential clear bit (with register specification)	α_1 = ... α_n = 0 (CY)	CY <- 0 , α_n <- 0 , ... , α_1 <- 0

4.2 Assignment Statements

Assignment statements Assign (=)

(1) Assign (=)

[Description format]

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

[Function]

- When there is no register specification
  β values on the right side are sequentially assigned to the left side.
- When there is a register specification
  β values on the right side are assigned to the specified register or to CY and their contents are sequentially assigned to the left side.

[Description]

- α and β are values that can be entered via the MOV or MOVW instruction.
  Up to 32 of the assignment operator “=” can be entered in one line. An error occurs when more than 32 are entered. If even one error occurs during sequential assignments , no instructions will be generated.

[Generated instructions]

(1) When there is no register specification

```
MOV     α 1 , β
```

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

(2) When there is no register specification and a sequential assignment is entered

```
MOV     α n , β
MOV     α n - 1 , β
        :
MOV     α 2 , β
MOV     α 1 , β
```

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

(3) When there is a register specification

```
MOV     Specified register , β
MOV     α 1 , Specified register
```

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

(4) When there is a register specification and a sequential assignment is entered

```
MOV     Specified register , β
MOV     α n , specified register
MOV     α n - 1 , specified register
        :
MOV     α 2 , specified register
MOV     α 1 , specified register
```

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

For details of combinations of α_n and β , see Table 4-5. Depending on the entered statement , α_n and β , indicates the specified register.

[Use examples]

(1) When there is no register specification

Outpu source	Input source
MOV1 CY , P1.1	CY = P1.1
MOV A , #4H	A = #4H
MOVW AX , SYMP	AX = SYMP
MOVW SP , #4FFFFH	SP = #4FFFFH
MOV DAT3 , A	DAT1 = DAT2 = DAT3 = A
MOV DAT2 , A	
MOV DAT1 , A	
MOVW DATA3P , AX	DATA1P = DATA2P = DATA3P = AX
MOVW DATA2P , AX	
MOVW DATA1P , AX	

(2) When there is a register specification

Outpu source	Input source
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	
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	

Table 4-5 Generated Instructions for Assignments

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α_n	a	CY		*3	*4						*3										
	b	Bit symbol	*3		*5																
	c	Byte user symbol	*6		*5		*1						*2								*1
	d	Byte data					*1														*1
	e	A			*1	*1		*1	*1	*1	*1							*1	*1	*1	*1
	f	Byte register					*1														*1
	g	sfr					*1														*1
	h	PSW					*1														*1
	i	Word user symbol	*3		*5		*1						*2								*2
	j	Word data											*2								*2
	k	AX			*2						*2	*2		*2							*2
	l	BC , DE , HL											*2								*2
	m	RP0 , RP1 , RP2 , RP3																			*2
	n	sfrp											*2								*2
	o	SP											*2								*2
	p	Direct access symbol					*1						*2								
	q	Indirect access symbol					*1														
	r	[DE]					*1														
	s	Immediate symbol																			

*1 :	Generates MOV instruction
*2 :	Generates MOVW instruction
*3 :	Generates MOV1 instruction
*4 :	Generates SET1 instruction when “1” has been entered as b. Generates CLR1 instruction when “0” has been entered. Generates MOV when any value other than “0” or “1” has been entered.
*5 :	Generates SET1 when “1” has been entered as b. Generates CLR1 when “0” has been entered.
*6 :	Generates MOV1 when any value other than "0" or "1" has been entered as an

Empty spaces indicate errors.

Assignment statements IncrementAssign (+=)

(2) IncrementAssign (+=)

[Description format]

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

[Function]

- When there is no register specification
  The two operands α and β are added and the result is assigned to α.
- When there is a register specification
  α is assigned to the specified register.
  The contents of the specified register are added to β and their result is assigned to the specified register.
  The contents of the specified register are assigned to α.
- Increment with carry ; no register specification
  An increment with carry operation is performed using the two operands α and β , and the result is assigned to α.
- Increment with carry ; with register specification
  The contents of α are assigned to the specified register.
  An increment with carry operation is performed using the contents of the specified register and β , and the result is assigned to the specified register.
  The contents of the specified register are assigned to α.

[Description]

- When there is no register specification
  The contents of α and β can be entered in ADD and ADDW.
- When there is a register specification
  The contents of α can be entered in MOV and MOVW.
  The contents of β can be entered in ADD and ADDW.
- Increment with carry ; no register specification
  The contents of α and β can be entered in ADDC.
- Increment with carry ; with register specification
  The contents of α can be entered in MOV.
  The contents of β can be entered in ADDC.

[Generated instructions]

(1) When there is no register specification

```
ADD     α , β
```

ADDW may be generated instead , depending on the operand.

(2) When there is a register specification

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

ADDW may be generated instead , depending on the operand.

(3) Increment with carry ; no register specification

```
ADDC    α , β
```

(4) Increment with carry ; with register specification

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

For details of combinations of α and β , see Table 4-6. Depending on the entered statement , α indicates the specified register.

[Use examples]

(1) When there is no register specification

Output source	Input source

ADD A , #0C0H
ADDW AX , #0C00H
	A += #0C0H
	AX += #0C00H

(2) When there is a register specification

Output source	Input source
MOV A , !ABC ADD A , #0FCH MOV !ABC , A	ABC += #0FCH (A)
MOVW AX , HL ADDW AX , #0FFFH MOVW HL , AX	HL += #0FFFH (AX)

(3) Increment with carry ; no register specification

Output source	Input source
ADDC A , #50H	A += #50H , CY

(4) Increment with carry ; with register specification

Output source	Input source
MOV A , PSW ADDC A , #50H MOV PSW , A	PSW += #50H , CY (A)

Table 4-6 Generated Instructions for Increment Assignments

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY																			
	b	Bit symbol																			
	c	Byte user symbol																			*1
	d	Byte data																			*1
	e	A			*1	*1	*1	*1			*1							*1	*1		*1
	f	Byte register					*1														
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX																			*2
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

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

*2 : Generates ADDW instruction.

Empty spaces indicate errors.

Assignment statements DecrementAssign (-=)

(3) DecrementAssign (-=)

[Description format]

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

[Function]

- When there is no register specification
  β is subtracted from α and the result is assigned to α.
- When there is a register specification
  α is assigned to the specified register.
  β is subtracted from the contents of the specified register and the result is assigned to the specified register.
  The contents of the specified register are assigned to α.
- Decrement with carry ; no register specification
  A decrement with carry operation is performed using the two operands α and β , and the result is assigned to α.
- Decrement with carry ; with register specification
  The contents of α are assigned to the specified register.
  An decrement with carry operation is performed using the contents of the specified register and β , and the result is assigned to the specified register.
  The contents of the specified register are assigned to α.

[Description]

- When there is no register specification
  The contents of α and β can be entered in SUB and SUBW.
- When there is a register specification
  The contents of α can be entered in MOV and MOVW.
  The contents of β can be entered in SUB and SUBW.
- Decrement with carry ; no register specification
  The contents of α and β can be entered in SUBC.
- Decrement with carry ; with register specification
  The contents of α can be entered in MOV.
  The contents of β can be entered in SUBC.

[Generated instructions]

(1) When there is no register specification

```
SUB     α , β
```

SUBW may be generated instead , depending on the operand.

(2) When there is a register specification

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

SUBW may be generated instead , depending on the operand.

(3) Decrement with carry ; no register specification

```
SUBC    α , β
```

(4) Decrement with carry ; with register specification

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

For details of combinations of α and β , see Table 4-7. Depending on the entered statement , α indicates the specified register.

[Use examples]

(1) When there is no register specification

Output source	Input source

SUB A , #0C0H
SUBW AX , #0C00H
	A -= #0C0H
	AX -= #0C00H

(2) When there is a register specification

Output source	Input source
MOV A , !ABC SUB A , #0FCH MOV !ABC , A	!ABC -= #0FCH (A)
MOVW AX , HL SUBW AX , #0FFFH MOVW HL , AX	HL -= #0FFFH (AX)

(3) Decrement with carry ; no register specification

Output source	Input source
SUBC A , #50H	A -= #50H , CY

(4) Decrement with carry ; with register specification

Output source	Input source
MOV A , PSW SUBC A , #50H MOV PSW , A	PSW -= #50H , CY (A)

Table 4-7 Generated Instructions for Decrement Assignments

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY																			
	b	Bit symbol																			
	c	Byte user symbol																			*1
	d	Byte data																			*1
	e	A			*1	*1	*1	*1			*1							*1	*1		*1
	f	Byte register					*1														
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX																			*2
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

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

*2 : Generates SUBW instruction.

Empty spaces indicate errors.

Assignment statements MultiplicationAssign (*=)

(4) MultiplicationAssign (*=)

[Description format]

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

[Function]

- When there is no register specification

  The contents of α are multiplied by β and the result is assigned to α.

- When there is a register specification

  The contents of α are assigned to the specified register.

  The contents of the specified register are multiplied by β and the result is assigned to the specified register.

  The contents of the specified register are assigned to α.

[Description]

- Where there is no register specification

  The contents of α can be entered in A , AX , or RP0.

  The contents of β can be entered in X only.

- Where there is a register specification

  A , AX , or RP0 can be entered as the specified register.

  The contents of α can be entered in a MOV or MOVW instruction.

  The contents of β can be entered in X only.

[Generated instructions]

(1) When there is no register specification

```
MULU    X
```

(2) When there is a register specification

```
MOVW   Specified register , α
MULU    X
MOVW   α , specified register
```

[Use examples]

(1) When there is no register specification

Output source	Input source
MULU X MULU X MULU X	A *= X AX *= X RP0 *= X

(2) When there is a register specification

Output source	Input source
MOV A , DATA MULU X MOV DATA , A	DATA *= X (A)
MOVW AX , DATAP MULU X MOVW DATAP , AX	DATAP *= X (AX)
MOVW RP0 , #30 MULU X	#30 *= X (RP0)

Assignment statements DivisionAssign (/=)

(5) DivisionAssign (/=)

[Description format]

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

[Function]

- When there is no register specification

  The contents of α are divided by β and the result is assigned to α.
- When there is a register specification

  The contents of α are assigned to the specified register.

  The contents of the specified register are divided by β and the result is assigned to the specified register.

  The contents of the specified register are assigned to α.

[Description]

- Where there is no register specification

  The contents of α can be entered in AX , or RP0.

  The contents of β can be entered in C only.
- Where there is a register specification

  AX , or RP0 can be entered as the specified register.

  The contents of α can be entered in a MOVW instruction.

  The contents of β can be entered in C only.

[Generated instructions]

(1) When there is no register specification

```
DIVUW  C
```

(2) When there is a register specification

```
MOVW   Specified register , α
DIVUW  C
MOVW   α , specified register
```

[Use examples]

(1) When there is no register specification

Output source	Input source
DIVUW C DIVUW C	AX /= C RP0 /= C

(2) When there is a register specification

Output source	Input source
MOVW AX , DATAP DIVUW C MOVW DATAP , AX	DATAP /= C (AX)
MOVW RP0 , #30 DIVUW C	#30 /= C (RP0)

(3) When there is a register specification

Output source	Input source
MOV A , CCV ROL A , 1 ROL A , 1 ROL A , 1 ROL A , 1 AND A , #LOW (0FFH SHL 4) MOV CCV , A	CCV <<= 4 (A)

Assignment statements LogicalANDAssign (&=)

(6) LogicalANDAssign (&=)

[Description format]

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

[Function]

- When there is no register specification

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

- When there is a register specification

  α is assigned to the specified register.

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

  The contents of the specified register are assigned to α.

[Description]

- Where there is no register specification

  The contents of α and β can be entered in AND and AND1.

- Where there is a register specification

  The contents of α can be entered in MOV and MOV1.

  The contents of β can be entered in AND and AND1.

[Generated instructions]

(1) When there is no register specification

< When α is CY >

```
AND1    CY , β
```

< When α is not CY >

```
AND     α , β
```

(2) When there is a register specification

< When the specified register is CY >

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

< When the specified register is not CY >

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

For details of combinations of α and β , see Table 4-8.

[Use examples]

(1) When there is no register specification

Output source	Input source
AND1 CY , P1S.1 AND A , #0FFH	CY &= P1S.1 A &= #0FFH

(2) When there is a register specification

Output source	Input source
MOV1 CY , A.1 AND1 CY , PORT3.0 MOV1 A.1, CY MOV A , [DE] AND A , #07H MOV [DE] , A	A.1 &= PORT3.0 (CY) [DE] &= #07H (A)

Table 4-8 Generated Instructions for Logical AND Assignments

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY		*2	*2						*2										
	b	Bit symbol																			
	c	Byte user symbol																			*1
	d	Byte data																			*1
	e	A			*1	*1	*1	*1			*1							*1	*1		*1
	f	Byte register					*1														
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX																			
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

*1 : Generates AND instruction.

*2 : Generates AND1 instruction.

Empty spaces indicate errors.

Assignment statements LogicalORAssign (|=)

(7) LogicalORAssign (|=)

[Description format]

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

[Function]

- When there is no register specification

  The logical OR (α **|** β) is obtained from the bits in α and β , and the result is assigned to α.

- When there is a register specification

  α is assigned to the specified register.

  The logical OR (specified register **|** β) is obtained from the bits in the specified register and β , and the result is assigned to the specified register.

  The contents of the specified register are assigned to α.

[Description]

- When there is no register specification

  The contents of α and β can be entered in OR and OR1.

- When there is a register specification

  The contents of α can be entered in MOV and MOV1.

  The contents of β can be entered in OR and OR1.

[Generated instructions]

(1) When there is no register specification

< When α is CY >

```
OR1     CY , β
```

< When α is not CY >

```
OR      α , β
```

(2) When there is a register specification

< When the specified register is CY >

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

< When the specified register is not CY >

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

For details of combinations of α and β , see Table 4-9.

[Use examples]

(1) When there is no register specification

Output source	Input source
OR1 CY , P1S.1	CY |= P1S.1
OR A , #0FFH	A |= #0FFH

(2) When there is a register specification

Output source	Input source
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	

Table 4-9 Generated Instructions for Logical OR Assignments

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY		*2	*2						*2										
	b	Bit symbol																			
	c	Byte user symbol																			*1
	d	Byte data																			*1
	e	A			*1	*1	*1	*1			*1							*1	*1		*1
	f	Byte register					*1														
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX																			
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

*1 :	Generates OR instruction.
*2 :	Generates OR1 instruction.

Empty spaces indicate errors.

Assignment statements LogicalXORAssign (^=)

(8) LogicalXORAssign (^=)

[Description format]

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

[Function]

- When there is no register specification

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

- When there is a register specification

  α is assigned to the specified register.

  The logical XOR (specified register ^ β) is obtained from the bits in the specified register and β , and the result is assigned to the specified register.

  The contents of the specified register are assigned to α.

[Description]

- When there is no register specification

  The contents of α and β can be entered in XOR and XOR1.

- When there is a register specification

  The contents of α can be entered in MOV and MOV1.

  The contents of β can be entered in XOR and XOR1.

[Generated instructions]

(1) When there is no register specification

< When α is CY >

```
XOR1    CY , β
```

< When α is not CY >

```
XOR     α , β
```

(2) When there is a register specification

< When the specified register is CY >

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

< When the specified register is not CY >

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

For details of combinations of α and β , see Table 4-10.

[Use examples]

(1) When there is no register specification

Output source	Input source
XOR1 CY , P1S.1 XOR A , #0FFH	CY ^= P1S.1 A ^= #0FFH

(2) When there is a register specification

Output source	Input source
MOV1 CY , A.1 XOR1 CY , PORT3.0 MOV1 A.1 , CY	A.1 ^= PORT3.0 (CY)
MOV A , [DE] XOR A , #07H MOV [DE] , A	[DE] ^= #07H (A)

Table 4-10 Generated Instructions for Logical XOR Assignments

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY		*2	*2						*2										
	b	Bit symbol																			
	c	Byte user symbol																			*1
	d	Byte data																			*1
	e	A			*1	*1	*1	*1			*1							*1	*1		*1
	f	Byte register					*1														
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX																			
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

*1 : Generates XOR instruction.

*2 : Generates XOR1 instruction.

Empty spaces indicate errors.

Assignment statements RightShiftAssign (>>=)

(9) RightShiftAssign (>>=)

[Description format]

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

[Function]

- When there is no register specification
  α is shifted to the right of the β bit , and the result is assigned to α.
- When there is a register specification
  α is assigned to the specified register.
  The contents of the specified register are shifted to the right of the β bit , and the result is assigned to the specified register.
  The contents of the specified register are assigned to α.

[Description]

- When there is no register specification
  The contents of α can be entered in A only.
  The contents of β can be entered as numerals from 1 to 7.
- When there is a register specification
  The contents of α can be entered in MOV.
  The contents of β can be entered as numerals from 1 to 7.
  The specified register can be entered in A only.

[Generated instructions]

(1) When there is no register specification
An AND instruction is generated after a ROR instruction is output β times.

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

(2) When there is a register specification

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

[Use examples]

(1) When there is no register specification

Output source	Input source
ROR A , 1 ROR A , 1 ROR A , 1 ROR A , 1 AND A , #0FFH SHR 4	A >>= 4

(2) When there is a register specification

Output source	Input source
MOV A , CCV ROR A , 1 ROR A , 1 ROR A , 1 ROR A , 1 AND A , #0FFH SHR 4 MOV CCV , A	CCV >>= 4 (A)

Assignment statements LeftShiftAssign (<<=)

(10) LeftShiftAssign (<<=)

[Description format]

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

[Function]

- When there is no register specification
  α is shifted to the left of the β bit , and the result is assigned to α.
- When there is a register specification
  α is assigned to the specified register.
  The contents of the specified register are shifted to the left of the β bit , and the result is assigned to the specified register.
  The contents of the specified register are assigned to α.

[Description]

- When there is no register specification
  The contents of α can be entered in A only.
  The contents of β can be entered as numerals from 1 to 7.
- When there is a register specification
  The contents of α can be entered in MOV.
  The contents of β can be entered as numerals from 1 to 7.
  The specified register can be entered in A only.

[Generated instructions]

(1) When there is no register specification
An AND instruction is generated after a ROL instruction is output β times.

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

(2) When there is a register specification

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

[Use examples]

(1) When there is no register specification

Output source	Input source
ROL A , 1 ROL A , 1 ROL A , 1 ROL A , 1 AND A , #LOW (0FFH SHL 4)	A <<= 4

(2) When there is a register specification

Output source	Input source
MOV A , CCV ROL A , 1 ROL A , 1 ROL A , 1 ROL A , 1 AND A , #LOW (0FFH SHL 4) MOV CCV , A	CCV <<= 4 (A)

4.3 Count Statements

Count statements Increment (++)

(11) Increment (++)

[Description format]

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

[Function]

- 1 is added to the contents of α.

[Description]

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

[Generated instructions]

```
INC     α
```

INCW may be generated depending on the operands.

For details of α , see Table 4-11.

[Use examples]

Output source	Input source
INC H INC CNT INCW HL	H++ CNT++ HL++

Table 4-11 Generated Instructions for Increment

Symbol			Generated instructions
α	a	CY	
	b	Bit symbol	
	c	Byte user symbol	*1
	d	Byte data	*1
	e	A	*1
	f	Byte register	*1
	g	sfr	
	h	PSW	
	i	Word user symbol	
	j	Word data	
	k	AX	*2
	l	BC , DE , HL	*2
	m	RP0 , RP1 , RP2 , RP3	*2
	n	sfrp	
	o	SP	
	p	Direct access symbol	
	q	Indirect access symbol	
	r	[DE]	
	s	Immediate symbol	

*1 : Generates INC instruction.

*2 : Generates INCW instruction.

Empty spaces indicate errors.

Count statements Decrement (--)

(12) Decrement (--)

[Description format]

```
[ Δ ] [ size specification ] [ Δ ] α [ Δ ] --
```

[Function]

- 1 is subtracted from the contents of α.

[Description]

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

[Generated instructions]

```
DEC     α
```

DECW may be generated depending on the operands.

For details of a , see Table 4-12.

[Use examples]

Output source	Input source
DEC H DEC CNT DECW HL	H-- CNT-- HL--

Table 4-12 Generated Instructions for Decrement

Symbol			Generated instructions
α	a	CY	
	b	Bit symbol	
	c	Byte user symbol	*1
	d	Byte data	*1
	e	A	*1
	f	Byte register	*1
	g	sfr	
	h	PSW	
	i	Word user symbol	
	j	Word data	
	k	AX	*2
	l	BC , DE , HL	*2
	m	RP0 , RP1 , RP2 , RP3	*2
	n	sfrp	
	o	SP	
	p	Direct access symbol	
	q	Indirect access symbol	
	r	[DE]	
	s	Immediate symbol	

*1 :	Generates DEC instruction.

*2 :	Generates DECW instruction.

Empty spaces indicate errors.

4.4 Exchange Statements

Exchange statements Exchange (<->)

(13) Exchange (<->)

[Description format]

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

[Function]

- When there is no register specification

  The contents of α and β are exchanged.

- When there is a register specification

  The contents of α are assigned to the specified register.

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

  The contents of the specified register are assigned to α.

[Description]

- Where there is no register specification

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

- When there is a register specification

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

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

[Generated instructions]

(1) When there is no register specification

```
XCH     α , β
```

XCHW may be generated depending on the operands.

(2) When there is a register specification

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

XCHW may be generated depending on the operands.

For details of combinations of α and β , see Table 4-13.

α indicates the specified register.

[Use examples]

(1) When there is no register specification

Output source	Input source
XCH A , B	A <-> B
XCHW AX , BC	AX <-> BC

(2) When there is a register specification

Output source	Input source
MOV A , DATA	DATA <-> B (A)
XCH A , B	
MOV DATA , A	
MOVW AX , DE	DE <-> BC (AX)
XCHW AX , BC	
MOVW DE , AX	

Table 4-13 Generated Instructions for Exchange

Symbol			β																		
			a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
α	a	CY																			
	b	Bit symbol																			
	c	Byte user symbol																			
	d	Byte data																			
	e	A			*1	*1		*1	*1		*1							*1	*1	*1	
	f	Byte register																			
	g	sfr																			
	h	PSW																			
	i	Word user symbol																			
	j	Word data																			
	k	AX												*2							
	l	BC , DE , HL																			
	m	RP0 , RP1 , RP2 , RP3																			
	n	sfrp																			
	o	SP																			
	p	Direct access symbol																			
	q	Indirect access symbol																			
	r	[DE]																			
	s	Immediate symbol																			

*1 : Generates XCH instructions.

*2 : Generates XCHW instructions.

Empty spaces indicate errors.

4.5 Bit Manipulation Statements

Bit manipulation statements Set bit (=)

(14) Set bit (=)

[Description format]

[Δ] α_1 [Δ] [= [Δ] α_2 [Δ] ...] = [Δ] 1 [Δ] [(CY specification)]
Enter a “1” at the end of the right side.

[Function]

- When there is no CY specification
  α_n is set (to a value of “1”).
- When there is a CY specification
  CY and α_n are set (to a value of “1”).

[Description]

- The contents of α_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 32 are entered. If even one error occurs during sequential assignments , no instructions will be generated.

[Generated instructions]

(1) When there is no CY specification

```
SET1    α 1
```

(2) When there is no CY specification in sequential assignments

```
SET1    α n
SET1    α n - 1
        :
SET1    α 2
SET1    α 1
```

(3) When there is a CY specification

```
SET1    CY
SET1    α 1
```

(4) When there is a CY specification in sequential assignments

```
SET1    CY
SET1    α n
SET1    α n - 1
        :
SET1    α 2
SET1    α 1
```

For details , see Table 4-14.

[Use examples]

(1) When there is no CY specification

Output source	Input source

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

(2) When there is a CY specification

Output source	Input source

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

Table 4-14 Generated Instructions for Set Bit

Symbol			Generated instructions
α	a	CY	*1
	b	Bit symbol	*1
	c	Byte user symbol	*1
	d	Byte data	
	e	A	
	f	Byte register	
	g	sfr	
	h	PSW	
	i	Word user symbol	*1
	j	Word data	
	k	AX	
	l	BC , DE , HL	
	m	RP0 , RP1 , RP2 , RP3	
	n	sfrp	
	o	SP	
	p	Direct access symbol	
	q	Indirect access symbol	
	r	[DE]	
	s	Immediate symbol	

*****1 : Generates SET1 instruction.

Empty spaces indicate errors.

Bit manipulation statements Clear bit (=)

(15) Clear bit (=)

[Description format]

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

[Function]

- When there is no CY specification
  α_n is cleared (to a value of “0”).
- When there is a CY specification
  CY and α_n are cleared (to a value of “0”).

[Description]

- The contents of α_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 32 are entered. If even one error occurs during sequential assignments , no instructions will be generated.

[Generated instructions]

(1) When there is no CY specification

```
CLR1    α 1
```

(2) When there is no CY specification in sequential assignments

```
CLR1    α n
CLR1    α n - 1
        :
CLR1    α 2
CLR1    α 1
```

(3) When there is a CY specification

```
CLR1    CY
CLR1    α1
```

(4) 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 4-15.

[Use examples]

(1) When there is no CY specification

Output source	Input source

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

(2) When there is a CY specification

Output source	Input source

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

Table 4-15 Generated Instructions for Clear Bit

Symbol			Generated instructions
α	a	CY	*1
	b	Bit symbol	*1
	c	Byte user symbol	*1
	d	Byte data	
	e	A	
	f	Byte register	
	g	sfr	
	h	PSW	
	i	Word user symbol	*1
	j	Word data	
	k	AX	
	l	BC , DE , HL	
	m	RP0 , RP1 , RP2 , RP3	
	n	sfrp	
	o	SP	
	p	Direct access symbol	
	q	Indirect access symbol	
	r	[DE]	
	s	Immediate symbol	

*****1 : Generates CLR1 instruction.

Empty spaces indicate errors.

CHAPTER 5   DIRECTIVES

This chapter describes directives. In this case , “directives” means various directives that the ST78K0 requires to execute a series of processes.

5.1 Overview of Directives

Directives are entered into source programs as various directives that the ST78K0 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 following pages describe the functions of the various directive functions.

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

Table 5-1 List of Directives

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

#DEFINE

(1) Symbol definition directive (#define)

[Description 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]

- The “#” character must always be entered at the start of the symbol , except when starting with a white space or a horizontal tab.
- Symbols start with a letter and are composed of alphanumeric characters. The first 31 characters are valid. If a symbol with more than 31 characters is specified, the 32nd and subsequent characters are ignored.
- 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.
- This directive is useful when coding easy-to-read symbols , such as numerical values.
- Reserved words cannot be entered as symbols.
- Reserved words can be entered as character strings.
- If the same symbol is defined twice , a warning message is output.
- Character strings that have been converted to secondary source files are output. The #define statement is not output.
- 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.
- This directive can be entered anywhere in the source code.
- A warning message is output when two or more symbols specifying option "-D" are entered , and the #define statement is valid.

[Use examples]

Output source	Input source
	#define TRUE 1
MOV X , #0	X = #0
CALL !xxx	CALL !xxx
MOV A , X	if (X == #TRUE) (A)
CMP A , #TRUE	
BNZ $?L1	
MOV B , #0C5H	B = #0C5H
?L1 :	endif

#IFDEF/#ELSE/#ENDIF

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

[Description format]

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

[Function]

- This directive performs conditional processing.
  (1) When the symbol has not been defined
  If #else has been entered , text 1 is skipped and text 2 becomes a processing object.
  (2) When the symbol has been defined
  If #else has been entered , text 1 becomes a processing object and text 2 is skipped.

[Description]

- The “#” character must always be entered at the start of the symbol , except when starting with a white space or a horizontal tab.
- Symbols start with a letter and are composed of alphanumeric characters. The first 31 characters are valid.
- Symbols are defined by a previously entered #define statement or by specifying the “-D” option at startup.
- This directive can be nested in up to eight levels.
- #else can be omitted.

[Use examples]

- When the following has been entered on the command line (and the symbol has been defined)
  C > st78k0 -c054 sample.st -dSYM

Output source	Input source

MOV A , #00H	#ifdef SYM
	A = #00H
	#else
	A = #0FFH
	#endif

- When the following has been entered on the command line (and the symbol has not been defined)

```
C > st780 -c054 sample.st
```

Output source	Input source
	#ifdef SYM
	A = #00H
	#else
MOV A , #0FFH	A = #0FFH
	#endif

#INCLUDE

(3) Include directive (#include)

[Description format]

```
[ Δ ] # [ Δ ] include Δ “file name”
```

[Function]

- This line is replaced by the specified file name and becomes a processing object as the ST78K0 source program.

[Description]

- The “#” character must always be entered at the start of the symbol , except when starting with a white space or a horizontal tab.
- This directive can be entered in any line in the source program.
- An include directive cannot be entered in an include file. In other words , nesting of include directives is not allowed.
- Input source file names specified at startup , output file names , and error file names cannot be specified as the file name in this directive.
- 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.
- The "-I" option can be used to specify a drive and directory for the include file when the ST78K0 is activated.

[Use examples]

Output source	Input source
	#include "sample.inc"
MOV A , #08H	A = SYM1 ; #define SYM1 #08H
MOV B , #0AH	B = SYM2 ; #define SYM2 #0AH

#DEFCALLT

(4) CALLT replacement directive (#defcallt)

[Description format]

```
[ Δ ] # [ Δ ] defcallt Δ CALLT table label
[ Δ ] CALL Δ ! 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]

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

[Use examples]

Output source			Input source		
			#DEFCALLT	@ABC	
				CALL	!ABC
			#ENDCALLT		
	MOV	R0 , #0		R0 = #0	
	CALLT	[@ABC]		CALL	!ABC
	CALL	!LABEL		CALL	!LABEL
CALL_T	CSEG	AT 40H	CALL_T	CSEG	AT 40H
@ABC :	DW	ABC	@ABC :	DW	ABC

CHAPTER 6   CONTROL INSTRUCTIONS

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

6.1 Overview of Control Instructions

Control instructions , which are entered into the source program , set various directives that the ST78K0 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 , $PC) , and kanji code specification control instructions ($KANJICODE) . For description of the description format for other control instructions , see the RA78K0 Assembler Package Language User’s Manual.

Table 6-1 lists control instructions that can be entered only in module headers.

Table 6-2 lists control instructions that are recognized as the module body.

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

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

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

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

6.3 Control Instruction Functions

The various functions of control instructions are listed in Table 6-3 below.

Table 6-3 Control Instruction List

Type of control instruction	Control instruction
Processor type specification instruction	$PROCESSOR
Kanji code specification control instructions	$KANJICODE

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

$PROCESSOR

(1) Processor type specification instruction ($PROCESSOR)

[Description format]

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

[Function]

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

[Description]

- 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.
- 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).
- An error occurs when this control instruction is entered more than once.
- An error occurs if neither this control instruction nor the “-C” option is used to specified a model name.
- An error occurs if this control instruction is entered anywhere other than in the module header.

[Code example]

```
$PROCESSOR ( 054 )
$PC ( 054 )
```

$KANJICODE

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

[Description format]

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

- Default assumption

Windows / HP-UX :	$KANJICODE SJIS
Sun OS :	$KANJICODE EUC

[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]

- This control instruction can be entered in the module header section of an input source file.
- An error occurs if this control instruction is entered anywhere other than in the module header.
- If this control instruction is entered more than once , the most recent one takes priority.
- This preprocessor outputs the specified control instruction to a secondary source file.

SJIS :	$KANJICODE SJIS
EUC :	$KANJICODE EUC
NONE :	$KANJICODE NONE

  If the same control instruction is entered in a secondary source file , the control instruction is not output. However , error checking is performed.
- Kanji code specifications are ranked in terms of priority as follows.

  1. Specification of -ZS/-ZE/-ZN option
  2. Specification of the kanji code specification control instruction ($KANJICODE)
  3. Specification of the environmental variable LANG78K
  4. Default specification of each OS

[Code example]

```
$KANJICODE SJIS
```

APPENDIX A SYNTAX LISTS

Table A-1 Control Statements

Control statement	Coding format

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

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

for statement
for ~ next
	for (expression ; conditional expression ; expression) [(register name)]
	    Instruction group
	next

while statement
while ~ endw
	while (conditional expression) [(register name)]
	    Instruction group
	endw

until statement
repeat ~ until
	repeat
	    Instruction group
	until (conditional expression) [(register name)]

break statement
break
	break

continue statement continue	continue
goto statement goto	goto label

Table A-1 Control Statements

Control statement	Coding format

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

while_bit statement
while_bit ~ endw
	while_bit (conditional expression) [(register name)]
	    Instruction group
	endw

until_bit statement
repeat ~ until_bit
	repeat
	    Instruction group
	until_bit (conditional expression) [(register name)]

Table A-2 Conditional Expressions

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

Table A-3 Expressions

	Expression	Coding format	Function
Assign (=)	Assign	$\alpha = \beta$	α <- β
	Assign (with register specification)	$\alpha = \beta (\gamma)$	(γ) <- β , α <- (γ)
	Sequential assign	$\alpha_1 = ... = \alpha_n = \beta$	α_1 <- β , ... , α_n <- β
	Sequential assign (with register specification)	$\alpha_1 = ... = \alpha_n = \beta (\gamma)$	γ <- β , α_1 <- γ , ... , α_n <- γ
IncrementAssign (+=)	Increment assignment	α += β	α <- $\alpha + \beta$
	Increment assignment (with register specification)	α += β (Register)	γ <- α , γ <- $\gamma + \beta$, α <- γ
	Increment assignment (with register specification)	α += β , CY	α <- $\alpha + \beta$, CY
	Increment assignment (with register specification)	α += β , CY (Register)	γ <- α , γ <- $\gamma + \beta$, CY , α <- γ
DecrementAssign (-=)	Decrement assignment	α -= β	α <- $\alpha - \beta$
	Decrement assignment (with register specification)	α -= β (Register)	γ <- α , γ <- $\gamma - \beta$, α <- γ
	Decrement assignment (with register specification)	α -= β , CY	α <- $\alpha - \beta$, CY
	Decrement assignment (with register specification)	α -= β , CY (Register)	γ <- α , γ <- $\gamma - \beta$, CY , α <- γ
MultiplicationAssign (*=)	Multiplication assignment	α *= β	α <- $\alpha * \beta$
	Multiplication assignment (with register specification)	α *= β (Register)	γ <- α , γ <- $\gamma * \beta$, α <- γ
DivisionAssign (/=)	Division assignment	α /= β	α <- α / β
	Division assignment (with register specification)	α /= β (Register)	γ <- α , γ <- γ / β , α <- γ
LogicalANDAssign (&=)	Logical AND assignment	α &= β	α <- $\alpha \cap \beta$
	Logical AND assignment (with register specification)	α &= β (Register)	γ <- α , γ <- $\gamma \cap \beta$, α <- γ
LogicalORAssign (|=)	Logical OR assignment	α |= β	α <- α U β
	Logical OR assignment (with register specification)	α |= β (Register)	γ <- α , γ <- γ U β , α <- γ
LogicalXORAssign (^=)	Logical XOR assignment	α ^= β	α <- α ^ β
	Logical XOR assignment (with register specification)	α ^= β (Register)	γ <- α , γ <- γ ^ β , α <- γ
RightShiftAssign (>>=)	Right shift (rotate) assignment	α >>= β	(α shifted to right of β bit)
	Right shift assignment (with register specification)	α >>= β (Register)	γ <- α , (γ shifted to right of β bit) , α <- γ

Table A-3 Expressions

	Expression	Coding format	Function
LeftShiftAssign (<<=)	Left shift assignment	α <<= β	(α shifted to left of β bit)
	Left shift assignment (with register specification)	α <<= β (Register)	γ <- α , (γ shifted to left of β bit) , α <- γ
Increment (++)	Increment	α ++	α <- α + 1
Decrement (--)	Decrement	α --	α <- α - 1
Exchange (<->)	Exchange	α <->= β	α <- α <->= β
	Exchange (with register specification)	α <->= β (γ)	γ <- α , γ <- γ <-> β , α <- γ
Set bit (=)	Set bit	α = 1	α <- 1
	Set bit (with register specification)	α = 1 (CY)	CY <- 1 , α <- 1
	Sequential set bit	α_1 = ... = α_n = 1	α_n <- 1 , ... , α_1 <- 1
	Sequential set bit (with register specification)	α_1 = ... = α_n = 1 (CY)	CY <- 1 , α_n <- 1 , ... , α_1 <- 1
Clear bit (=)	Clear bit	α = 0	α <- 0
	Clear bit (with register specification)	α = 0 (CY)	CY <- 0 , α <- 0
	Sequential clear bit	α_1 = ... = α_n = 0	α_n <- 0 , ... , α_1 <- 0
	Sequential clear bit (with register specification)	α_1 = ... = α_n = 0 (CY)	CY <- 0 , α_n <- 0 , ... , α_1 <- 0

Table A-4 Directives

Directive	Coding format
#define Symbol definition directive (#define)	#define symbol character string
#ifdef Conditional processing directive (#ifdef/#else/#endif)	#ifdef symbol text 1 #else text 2 #endif
#include Include directive (#include)	#include "file name"
#defcallt CALLT replacement directive (#defcallt)	#defcallt CALLT table label CALL label #endcallt

Table A-5 Control Instructions

Control Instruction	Coding format
Processor type specification instruction ($PROCESSOR)	$PROCESSOR (model name)
Kanji code specification control instruction ($KANJICODE)	$KANJICODE Kanji code

APPENDIX B LISTS OF GENERATED INSTRUCTIONS

Table B-1 Generated Instructions for Comparison Expressions

Comparison expression		Generated instruction	Control statement condition
Equal (==)	α == β	CMP (W) α , β BNZ $?LFALSE	lower case letters
		CMP (W) α , β BZ $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α == β (γ)	MOV (W) γ , α CMP (W) γ , β BNZ $?LFALSE	lower case letters
		MOV (W) γ , α CMP (W) γ , β BZ $?LTRUE BR LFALSE ?LTRUE :	upper case letters
NotEqual (!=)	α != β	CMP (W) α , β BZ $?LFALSE	lower case letters
		CMP (W) α , β BNZ $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α != β (γ)	MOV (W) γ , α CMP (W) γ , β BZ $?LFALSE	lower case letters
		MOV(W) γ , α CMP(W) γ , β BNZ $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters

Table B-1 Generated Instructions for Comparison Expressions

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

Table B-1 Generated Instructions for Comparison Expressions

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

γ :   specified register

Table B-2 Generated Instructions for Test Bit Expressions

Test bit expression	Generated instruction	Control statement condition
Bit symbol if_bit (bit symbol) elseif_bit (bit symbol) while_bit (bit symbol) until_bit (bit symbol)	BNC $?LFALSE	lower case letters (CY)
	BNZ $?LFALSE	lower case letters (Z)
	BF bit symbol , $?LFALSE	lower case letters
	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
!bit symbol if_bit (!bit symbol) elseif_bit (!bit symbol) while_bit (!bit symbol) until_bit (!bit symbol)	BC $?LFALSE	lower case letters (CY)
	BZ $?LFALSE	lower case letters (Z)
	BT bit symbol , $?LFALSE	lower case letters
	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

Logic expression		Generated instruction	Control statement condition
Logical AND (&&)	α == β &&	CMP (W) α , β BNZ $?LFALSE	lower case letters
		CMP (W) α , β BZ $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α != β &&	CMP (W) α , β BZ $?LFALSE	lower case letters
		CMP (W) α , β BNZ $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α < β &&	CMP (W) α , β BNC $?LFALSE	lower case letters
		CMP (W) α , β BC $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α > β &&	CMP (W) α , β BZ $?LFALSE BC $?LFALSE	lower case letters
		CMP (W) α , β BZ $$ + 4 BNC $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α >= β &&	CMP (W) α , β BC $?LFALSE	lower case letters
		CMP (W) α , β BNC $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	α <= β &&	CMP (W) α , β BZ $$ + 4 BNC $?LFALSE	lower case letters
		CMP (W) α , β BZ $?LTRUE BC $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters

Table B-3 Generated Instructions for Logic Expressions

Logic expression		Generated instruction	Control statement condition
Logical AND (&&)	CY &&	BNC $?LFALSE	lower case letters
		BC $?LTRUE BR ?LFALSE ?LTRUE :	upper case letters
	Z &&	BNZ $?LFALSE	lower case letters
		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

Table B-3 Generated Instructions for Logic Expressions

Logic expression		Generated instruction		Control statement condition
Logical OR (||)	α == β ||	CMP (W) BZ	α , β $?LFALSE	none
	α != β ||	CMP (W) BNZ	α , β $?LFALSE	
	α < β ||	CMP (W) BC	α , β $?LFALSE	
	α > β ||	CMP (W) BZ BNC	α , β $?LFALSE $?LFALSE	
	α >= β ||	CMP (W) BNC	α , β $?LFALSE	
	α <= β ||	CMP (W) BZ BC	α , β $?LFALSE $?LFALSE	
	CY ||	BC	$?LFALSE	
	Z ||	BZ	$?LFALSE	
	bit symbol ||	BT	bit symbol , $?LFALSE	
	!CY ||	BNC	$?LFALSE	
	!Z ||	BNZ	$?LFALSE	
	!bit symbol ||	BF	bit symbol , $?LFALSE	

Table B-4 Expressions

Expression		Generated instruction
Assign (=)	$\alpha = \beta$	MOV1 α_1 , β
		MOV α_1 , β
		MOVW α_1 , β
	$\alpha = \beta (\gamma)$	MOV1 γ , β MOV1 α_1 , γ
		MOV γ , β MOV α_1 , γ
		MOVW γ , β MOVW α_1 , γ
IncrementAssign (+=)	α += β	ADD α , β
		ADDW α , β
	α += $\beta (\gamma)$	MOV γ , α ADD γ , β MOV α , γ
		MOVW γ , α ADDW γ , β MOVW α , γ
	α += β , CY	ADDC α , β
	α += β , CY (γ)	MOV γ , α ADDC γ , β MOV α , γ
DecrementAssign (-=)	α -= β	SUB α , β
		SUBW α , β
	α -= $\beta (\gamma)$	MOV γ , α SUB γ , β MOV α , γ
		MOVW γ , α SUBW γ , β MOVW α , γ
	α -= β , CY	SUBC α , β
	α -= β , CY (γ)	MOV γ , α SUBC γ , β MOV α , γ
MultiplicationAssign (*=)	α *= β	MULUX X
	α *= $\beta (\gamma)$	MOVW γ , α MULU X MOVW α , γ

Table B-4 Expressions

Expression		Generated instruction
DivisionAssign (/=)	α /= β	DIVUM C
	α /= β (γ)	MOVW γ , α DIVUM C MOVW α , γ
LogicalANDAssign (&=)	α &= β	AND1 CY , β
		AND α , β
	α &= β (γ)	MOV1 γ , α AND1 γ , β MOV1 α , γ
		MOV γ , α AND γ , β MOV α , γ
LogicalORAssign (|=)	α |= β	OR1 CY , β
		OR α , β
	α |= β (γ)	MOV1 γ , α OR1 γ , β MOV1 α , γ
		MOV γ , α OR γ , β MOV α , γ
LogicalXORAssign (^=)	α ^= β	XOR1 α , β
		XOR α , β
	α ^= β (γ)	MOV1 γ , α XOR1 γ , β MOV1 α , γ
		MOV γ , α XOR γ , β MOV α , γ
RightShiftAssign (>>=)	α >>= β	ROR A , 1 : AND A , #0FFH SHR β
	α >>= β (γ)	MOV A , α ROR A , 1 : AND A , #0FFH SHR β MOV α , A

Table B-4 Expressions

Expression		Generated instruction
LeftShift Assign (<<=)	α <<= β	ROL A , 1 : AND A , #LOW (0FFH SHL β)
	α <<= β (γ)	MOV A , α ROL A , 1 : AND A , #LOW (0FFH SHL β) MOV α , A
Increment (++)	α ++	INC α
		INCW α
Decrement (--)	α --	DEC α
		DECW α
Exchange (<->)	α <-> β	XCH α , β
		XCHW α , β
	α <-> β (γ)	MOV γ , α XCH γ , β MOV α , γ
		MOVW γ , α XCHW γ , β MOVW α , γ
Set bit (=)	α = 1	SET1 $α_1$
	α = 1 (CY)	SET1 CY SET1 $α_1$
Clear bit (=)	α = 0	CLR1 $α_1$
	α = 0 (CY)	CLR1 CY CLR1 $α_1$

APPENDIX C INDEX

Symbols
#DEFCALLT ... 143
#DEFINE ... 138
#ELSE ... 140
#ENDIF ... 140
#IFDEF ... 30, 140
#INCLUDE ... 30, 142
$KANJICODE ... 145, 149
$PC ... 145
$PROCESSOR ... 145, 148

A
Assembler control instruction ... 24, 29
Assembler operator ... 23
Assembly language ... 18
Assign ... 99
Assignment ... 96, 103, 106, 109, 111, 113, 115, 117, 120, 122
Assignment statement ... 96, 99

B
Bit manipulation statement ... 96, 98, 130
break ... 56
Byte access ... 22
Byte symbol ... 17

C
CALLT replacement directive ... 143
Character set ... 20
Clear bit ... 133
Comment ... 30
Comment statement ... 29
Comparison expression ... 60
Conditional branch ... 36
Conditional expression ... 18
Conditional loop ... 46
Conditional processing directive ... 140
Constant ... 22
continue ... 57
Control statement ... 18, 23, 30
Count statement ... 96, 98, 124

D
Decrement ... 126
DecrementAssign ... 106
Directive ... 23, 30, 136
DivisionAssign ... 111

E
Equal ... 62
Error message ... 29
Exchange ... 128
Exchange statement ... 96, 98, 128
Expression ... 22
Expression statement ... 18, 30

F
for ... 46
FOREVER (forever) ... 80

G
goto ... 58
GreaterEqual ... 74
GreaterThan ... 71

I
Identifier ... 22
if ... 36
if_bit ... 38
Illegal character ... 22, 28
Include directive ... 142
Increment ... 124
IncrementAssign ... 103
Invalid character ... 22

K
Kanji code specification control instruction ... 149

L
Label ... 17, 25
LeftShiftAssign ... 122
LessEqual ... 77
LessThan ... 68
Letter ... 20
Line feed ... 29
Logical AND ... 90
Logical operation ... 89
Logical OR ... 94
LogicalANDAssign ... 113
LogicalORAssign ... 115
LogicalXORAssign ... 117
Lower case ... 20
Lower case letter ... 31

M
Module body ... 145
Module header ... 29, 145
MultiplicationAssign ... 109

N
Negative logic (bit) ... 86
NotEqual ... 65

Numeral ... 20

O
Operator ... 23

P
Positive logic (bit) ... 83
Processor type specification instruction ... 148

R
Register ... 24
RightShiftAssign ... 120

S
Set bit ... 130
Source program ... 136, 144
Special character ... 20
ST78K0 ... 13
Structured assembler preprocessor ... 13
Structured assembly language ... 18
switch ... 41
Symbol ... 22
Symbol definition directive ... 138

T
Test bit expression ... 82

U
until ... 53
until_bit ... 55
Upper case ... 20
Upper case letter ... 31
User symbol ... 22

W
while ... 49
while_bit ... 51
Word access ... 22
Word symbol ... 17