To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

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

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

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

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

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

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

User's Manual

RA78K0S Ver. 2.00

Assembler Package

Language

Target Device
78K0S Microcontrollers

Document No. U17390EJ2V0UM00 (2nd edition)
Date Published August 2007

[MEMO]

Windows is either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Solaris and SunOS are trademarks of Sun Microsystems, Inc.

- **The information in this document is current as of August, 2007. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".

  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

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

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

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

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

(Note)

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

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

M8E 02.11-1

INTRODUCTION

This manual is intended to give users who wish to develop software using the RA78K0S an understanding of the functions of each program in the RA78K0S Microcontrollers Assembler Package (hereafter referred to as "the RA78K0S") and of the correct methods of using the package.

This manual does not cover the language, such as the expressions of assembler directives and source programs used in the RA78K0S. Therefore, before reading this manual, read the **RA78K0S Assembler Package User's Manual Operation (U17391E)** (hereafter referred to as "Operation").

The contents of this manual are intended for use with Ver. 1.50 of the RA78K0S.

[Target Readers]

The RA78K0S is intended for users who understand the functions and instructions of the microcontroller to be developed (78K0S Microcontrollers).

[Organization]

This manual consists of the following chapters and appendixes:

CHAPTER 1 GENERAL
Outlines all of the basic functions of the RA78K0S.

CHAPTER 2 HOW TO DESCRIBE SOURCE PROGRAMS
Describes the description methods, expressions and operators of the source program.

CHAPTER 3 DIRECTIVES
Explains how to write and use directives, including application examples.

CHAPTER 4 CONTROL INSTRUCTIONS
Explains how to write and use control instructions, including application examples.

CHAPTER 5 MACROS
Explains all macro functions, including macro definition, macro reference, and macro expansion.
Macro directives are explained in **CHAPTER 3 DIRECTIVES**.

CHAPTER 6 PRODUCT UTILIZATION
Introduces some measures recommended for describing a source program.

APPENDICES These contain a list of reserved words, a list of directives, and an index.

The instruction sets are not detailed in this manual. For these instructions, refer to the user's manual of the microcontroller (instruction version) 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 assembler for the first time are encouraged to read from **CHAPTER 1 GENERAL** of this manual. Those who have a general knowledge of assembler programs may skip **CHAPTER 1 GENERAL** of this manual. However, be sure to read **1.2 Reminders Before Program Development** and **CHAPTER 2 HOW TO DESCRIBE SOURCE PROGRAMS**.

Those who wish to know the directives and control instructions of the assembler are encouraged to read **CHAPTERS 3 DIRECTIVES** and **4 CONTROL INSTRUCTIONS**, respectively. The format, function, use, and application examples of each directive or control instruction are detailed in these chapters.

[Conventions]

The following symbols and abbreviations are used throughout this manual:

⋮:	Indicates that the same expression is repeated.
[ ]:	Item(s) in brackets can be omitted.
' ':	Characters enclosed in ' ' (quotation marks) will be listed as they appear.
" ":	Characters enclosed in " " (double quotation marks) are titles of chapters, paragraphs, sections, diagrams or tables to which the reader is asked to refer.
___:	Indicates an important point, or characters that are to be input in a usage example.
□:	Indicates one blank space.
∆:	Indicates one or more blank or TAB.
∇:	Indicates zero or more blanks or TABs (i.e. blanks may be omitted).
/:	Indicates a break between characters.
~:	Indicates continuity.
[↵]:	Indicates pressing of the Return key.
Note:	Footnote for item marked with **Note** in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information

[Related Documents]

The documents 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 related to development tools (user's manuals)

Document Name		Document Number
CC78K0S Ver. 2.00 C Compiler	Operation	U17416E
	Language	U17415E
RA78K0S Ver. 2.00 Assembler Package	Operation	U17391E
	Language	This manual
	Structured Assembly Language	U17389E
SM+ System Simulator	Operation	U18601E
	User Open Interface	U18602E
SM78K Series Ver.2.52 System Simulator	Operation	U16768E
ID78K0S-NS Integrated Debugger Ver. 2.52	Operation	U16584E
ID78K0S-QB Integrated Debugger Ver. 3.00	Operation	U17287E
PM+ Ver. 6.30 Project Manager		U18416E

Caution The contents of the above related documents are subject to change without notice. Be sure to use the latest edition of each document when designing your system.

[MEMO]

CONTENTS

LIST OF FIGURES

Figure No. Title , Page

LIST OF TABLES

Table No. Title , Page

CHAPTER 1 GENERAL

This chapter describes the role of the RA78K0S in microcontroller software development and the features of the RA78K0S.

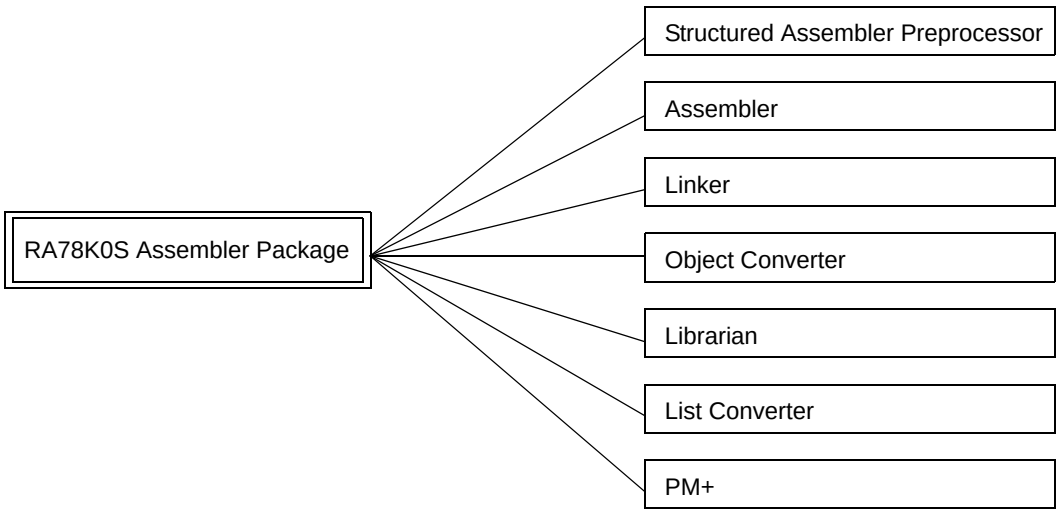
1.1 Assembler Overview

The RA78K0S Assembler Package (hereafter referred to as "the RA78K0S") is a generic term for a series of programs designed to translate source programs coded in the assembly language for 78K0S Series microcontrollers into machine language coding.

The RA78K0S contains six programs : Structured Assembler Preprocessor, Assembler, Linker, Object Converter, Librarian, and List Converter.

In addition, a PM+ that helps you perform a series of operations including editing, compiling / assembling, linking, and debugging your program on Windows is also supplied with the RA78K0S.

Figure 1-1 RA78K0S Assembler Package



1.1.1 What is an assembler?

(1) Assembly language and machine language

An assembly language is the most fundamental programming language for a microcontroller.

Programs and data are required for the microprocessor in a microcontroller to do its job. These programs and data must be written by users to the memory of the microcontroller. The programs and data handled by the microcontroller are collections of binary numbers called machine language. For users, however, machine language code is difficult to remember, causing errors to occur frequently. Fortunately, methods exist whereby English abbreviations or mnemonics are used to represent the meanings of the original machine language codes in a way that is easy for user to comprehend. The basic programming language system that uses this symbolic coding is called an assembly language.

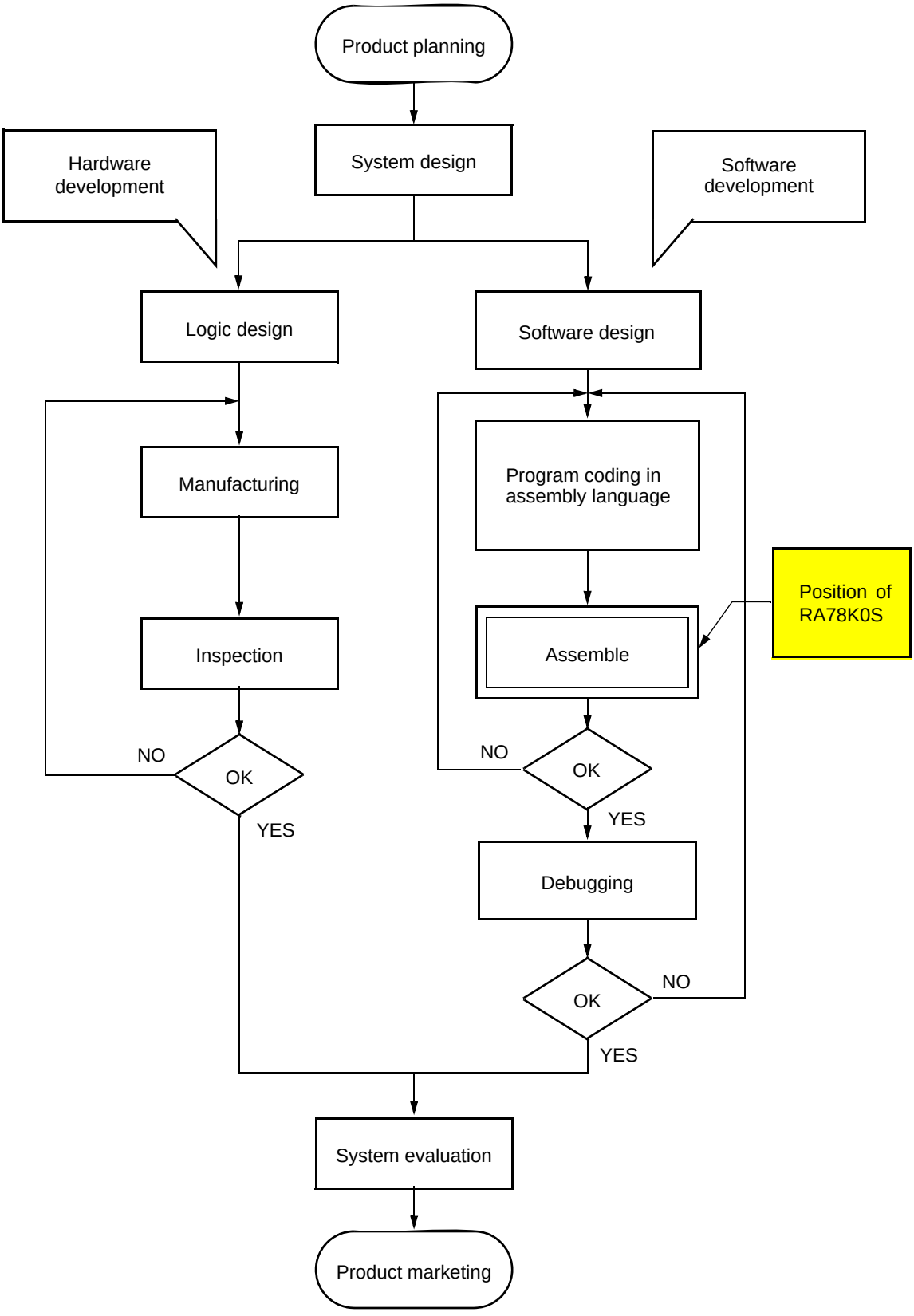
Since machine language is the only programming language in which a microcontroller can handle programs, however, another program is required that translates programs created in assembly language into machine language. This program is called an assembler.

Figure 1-2 Flow of Assembler

(2) Development of microcontroller-applied products and the role of RA78K0S

Figure 1-3 illustrates the position of "assemble in the product development process".

Figure 1-3 Development Process of Microcontroller-Applied Products



1.1.2 Relocatable assembler

The machine language translated from a source language by the assembler is written to the memory of the microcontroller before use. To do this, the location in memory where each machine language instruction is to be written must already be determined.

Therefore, information is added to the machine language assembled by the assembler, stating where in memory each machine language instruction is to be located.

Depending on the method of locating addresses to machine language instructions, assemblers can be broadly divided into "absolute assemblers" and "relocatable assemblers".

- Absolute assembler

  An absolute assembler locates machine language instructions assembled from the assembly language to absolute addresses.

- Relocatable assembler

  In a relocatable assembler, the addresses determined for the machine language instructions assembled from the assembly language are tentative. Absolute addresses are determined subsequently by the linker.

In the past, when a program was created with an absolute assembler, programmers had to, as a rule, complete programming at the same time. However, if all the components of a large program are created as a single entity, the program becomes complicated, making analysis and maintenance of the program difficult. To avoid this, such large programs are developed by dividing them into several subprograms, called modules, for each functional unit. This programming technique is called modular programming.

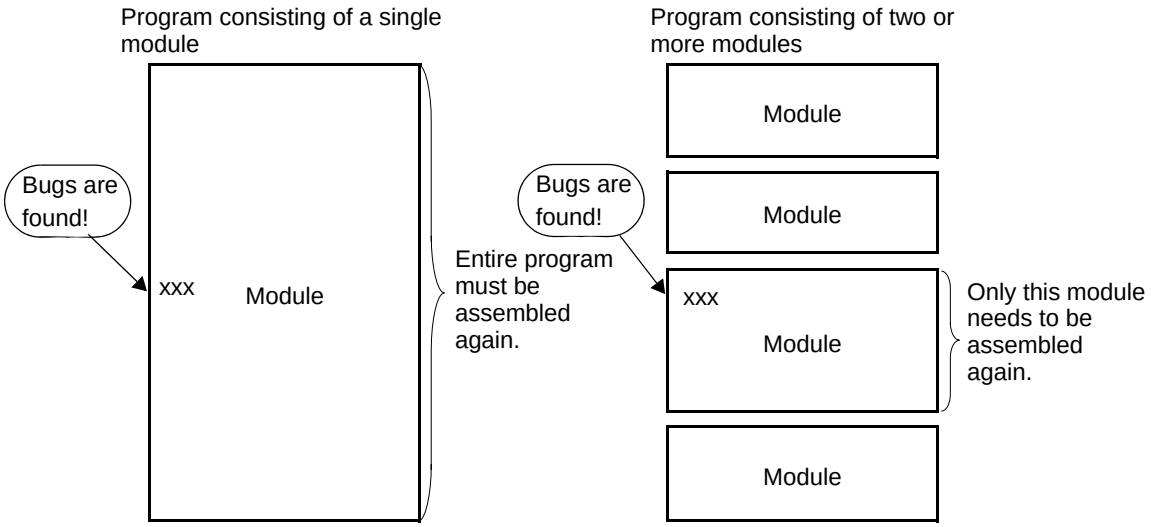
A relocatable assembler is an assembler suitable for modular programming, which has the following advantages:

(1) Increase in development efficiency

It is difficult to write a large program all at the same time. In such cases, dividing the program into modules for individual functions enables two or more programmers to develop subprograms in parallel to increase development efficiency.

Furthermore, if any bugs are found in the program, it is not necessary to assemble the entire program just to correct one part of the program ; just the module that must be corrected can be reassembled. This shortens the debugging time.

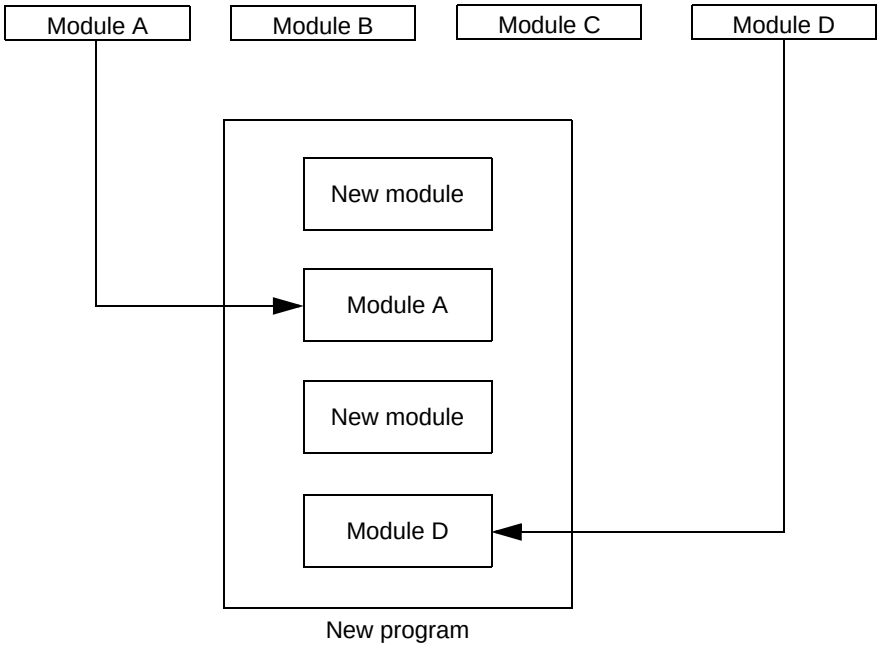
Figure 1-4 Reassembly for Debugging



(2) Utilization of resources

Highly reliable, highly versatile modules that have been previously created can be reused for the creation of another program. If you accumulate such high-versatility modules as software resources, you can save time and labor in developing a new program.

Figure 1-5 Program Development Using Existing Module



1.2 Reminders Before Program Development

Refer to the following before beginning program development.

1.2.1 Maximum performance characteristics of RA78K0S

(1) Maximum performance characteristics of assembler

Table 1-1 Maximum Performance Characteristics of Assembler

Item	Maximum Performance Characteristics	
	Windows version	UNIX version
Number of symbols (local + public)	65,535 symbols	65,535 symbols
Number of symbols for which cross-reference list can be output	65,534 symbols	65,534 symbols
Maximum size of macro body for one macro reference	1 M bytes	1 M bytes
Total size of all macro bodies	10 M bytes	10 M bytes
Number of segments in one file	256 segments	256 segments
Macro and include specifications in one file	10,000	10,000
Macro and include specifications in one include file	10,000	10,000
Relocation data[Note 1]	65,535 items	65,535 items
Line number data	65,535 items	65,535 items
Number of BR directives in one file	32,767 directives	32,767 directives
Number of characters per line	2,048 characters[Note 2]	2,048 characters[Note 2]
Symbol length	256 characters	256 characters
Number of definitions of switch name[Note 3]	1,000	1,000
Character length of switch name[Note 3]	31 characters	31 characters
Number of nesting levels on include file in one file	8 levels	8 levels

Notes 1. "Relocation data" is the data transferred to the linker when the assembler cannot decide the symbol values.

For example, when referring to an external reference symbol by a MOV instruction, two items of relocation data are generated in the .rel file.

Notes 2. This includes the carriage return and feed codes. If 2,049 characters or more are described on a line, a warning message is output and any characters at or over 2,049 are ignored.

Notes 3. Switch name is set to true or false by SET / RESET directives and used with $IF, etc.

(2) Maximum performance characteristics of linker

Table 1-2 Maximum Performance Characteristics of Linker

Item	Maximum Performance Characteristics	
	Windows version	UNIX version
Number of symbols (local + public)	65,535 symbols	65,535 symbols
Line number data of same segment	65,535 items	65,535 items
Number of segments	65,535 segments	65,535 segments
Number of input modules	1,024 modules	1,024 modules

1.3 Features of RA78K0S

The RA78K0S has the following features :

(1) Macro function

When the same group of instructions must be described in a source program over and over again, a macro can be defined by giving a single macro name to the group of instructions.

By using this macro function, coding efficiency and readability of the program can be increased.

(2) Optimize function of branch instructions

The RA78K0S has an Automatic Branch Instruction Selection Directive "BR (branch)".

To create a program with high memory efficiency, a byte branch instruction must be described according to the branch destination range of the branch instruction. However, it is troublesome for the programmer to describe a branch instruction by paying attention to the branch destination range for each branching. By describing the BR directive, the assembler generates the appropriate branch instruction according to the branch destination range. This is called the optimize function of branch instructions.

(3) Conditional assembly function

With this function, a part of a source program can be specified for assembly or non-assembly according to a predetermined condition.

If a debug statement is described in a source program, whether or not the debug statement should be translated into machine language can be selected by setting a switch for conditional assembly. When the debug statement is no longer required, the source program can be assembled without major modifications to the program.

CHAPTER 2 HOW TO DESCRIBE SOURCE PROGRAMS

This chapter describes the description methods, expressions and operators of the source program.

2.1 Basic Configuration

When a source program is described by dividing it into several modules, each module that becomes the unit of input to the assembler is called a source module (if a source program consists of a single module, "source program" means the same as "source module").

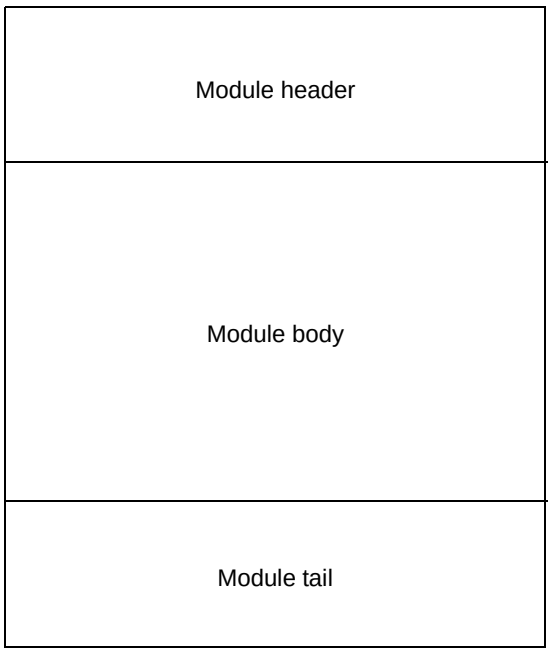
Each source module that becomes the unit of input to the assembler consists mainly of the following three parts :

(1) Module header

(2) Module body

(3) Module tail

Figure 2-1 Configuration of Source Module



2.1.1 Module header

In the module header, the control instructions shown in Table 2-1 below can be described. Note that these control instructions can only be described in the module header.

Also, the module header can be omitted.

Table 2-1 Instructions That Can Be Described in Module Header

Item That Can Be Described	Explanation	Chapter / Section in This Manual
Control instructions that have the same functions as assembler options	Control instructions that have the same functions as assembler options are as follows : - PROCESSOR - XREF / NOXREF - DEBUG / NODEBUG, DEBUGA / NODEBUGA - TITLE - SYMLIST / NOSYMLIST - FORMFEED / NOFORMFEED - WIDTH - LENGTH - TAB	CHAPTER 4 CONTROL INSTRUCTIONS
Special control instructions output by high-level programs such as C compiler and structured assembler preprocessor	Special control instructions output by high-level programs such as C compiler and structured assembler preprocessor are as follows : - TOL_INF - DGS - DGL	

2.1.2 Module body

In the module body, the following instructions cannot be described :

- Control instructions that have the same functions as assembler options

All other directives, control instructions, and instructions can be described in the module body.

The module body must be described by dividing it into units, called "segments".

The user may define the following four segments with a directive corresponding to each segment :

(1) Code segment

Must be defined with the CSEG directive.

(2) Data segment

Must be defined with the DSEG directive.

(3) Bit segment

Must be defined with the BSEG directive.

(4) Absolute segment

Must be defined by specifying a location address for the relocation attribute (AT location address) with the CSEG, DSEG, or BSEG directive. This segment may also be defined with the ORG directive.

The module body may be configured with any combination of segments.

However, a data segment and a bit segment should be defined before a code segment.

2.1.3 Module tail

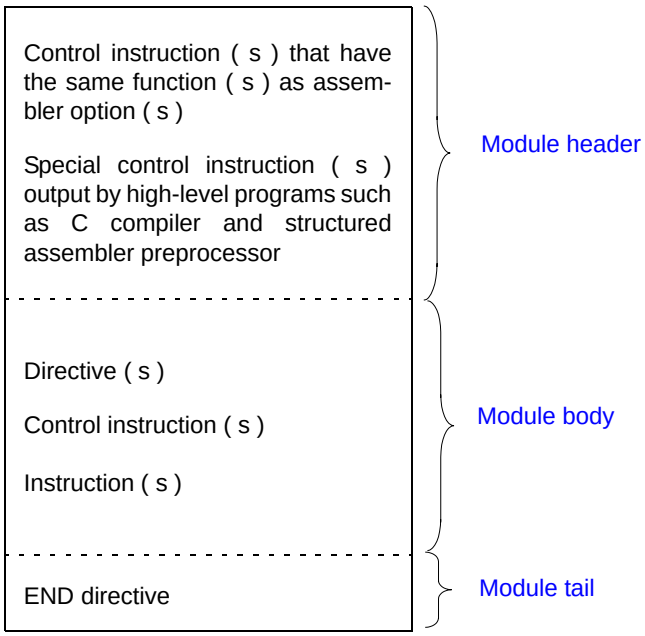
The module tail indicates the end of the source module. The END directive must be described in this part.

If anything other than a comment, a blank, a tab, or a line feed code is described following the END directive, the assembler will output a warning message and ignore the characters described after the END directive.

2.1.4 Overall configuration of source program

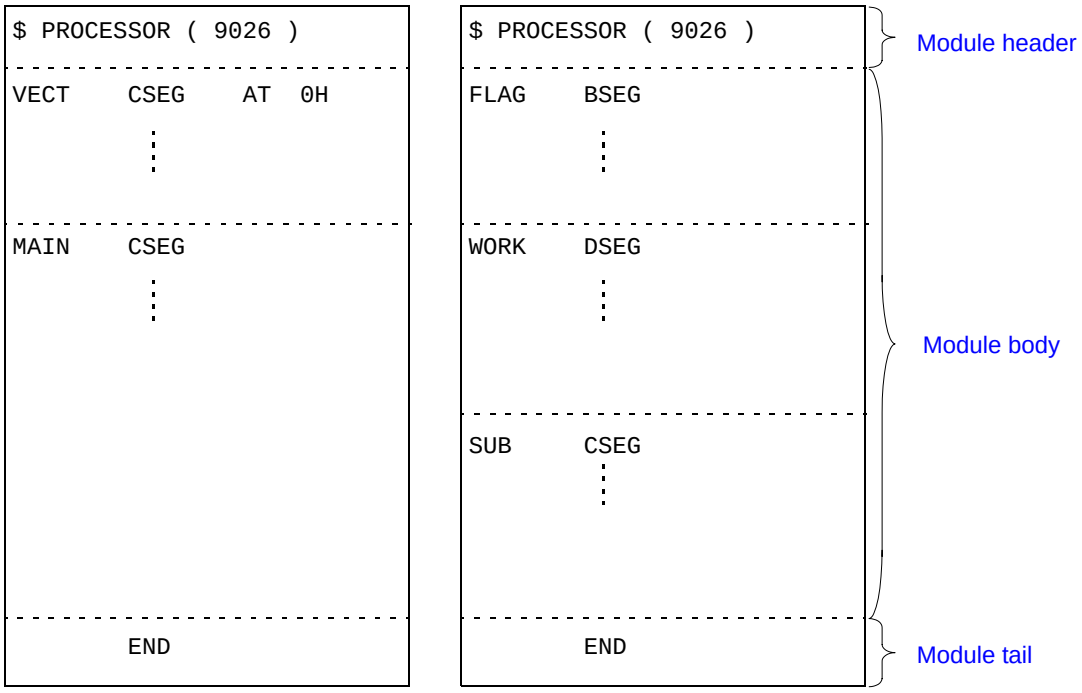
The overall configuration of a source module (source program) is as shown below.

Figure 2-2 Overall Configuration of Source Module



Examples of simple source module configurations are shown in Figure 2-3.

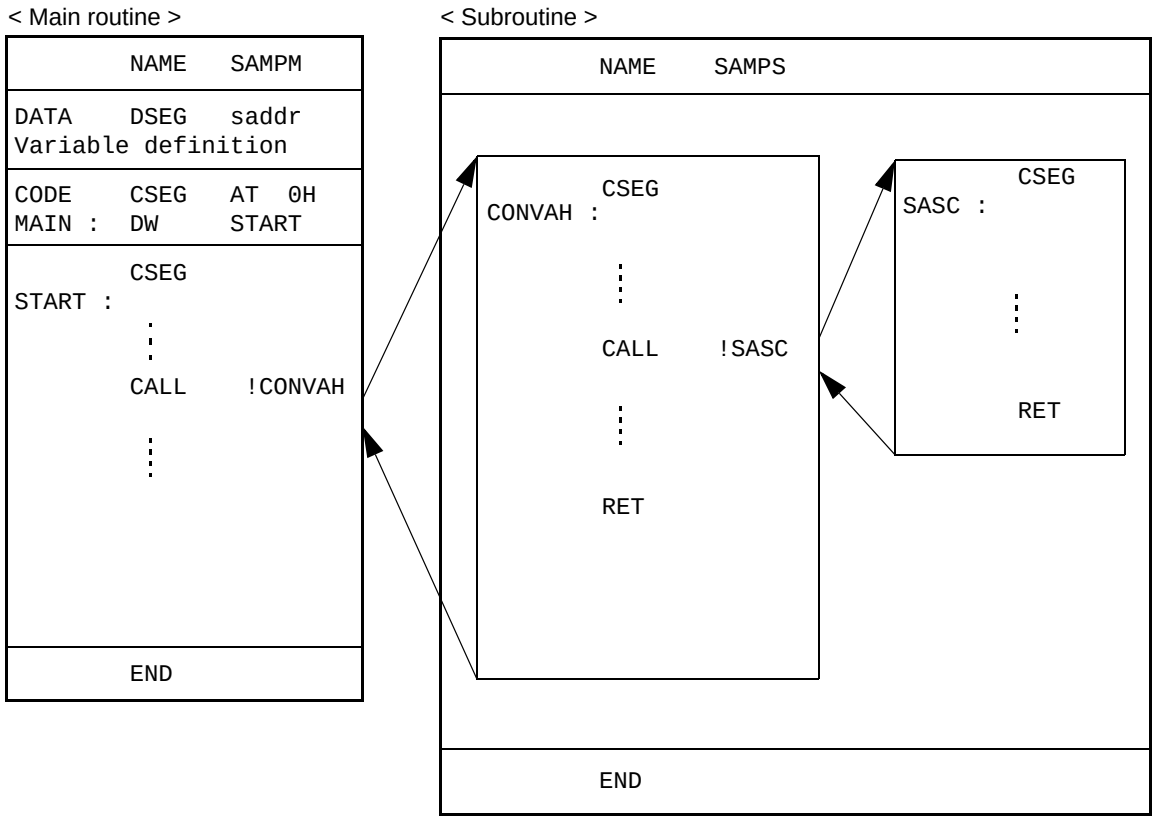
Figure 2-3 Examples of Source Module Configurations



2.1.5 Description example

In this subsection, a description example of a source module (source program) is shown as a sample program. The configuration of the sample program can be illustrated simply as follows.

Figure 2-4 Configuration of Sample Program



< Main routine >

```
        NAME    SAMPM           ; (1)
; ************************************
;       HEX -> ASCII Conversion Program
;       main-routine
; ************************************

        PUBLIC  MAIN , START    ; (2)
        EXTRN   CONVAH          ; (3)
        EXTRN   _@STBEG         ; (4)

DATA    DSEG    saddr           ; (5)
HDTSA : DS      1
STASC : DS      2

CODE    CSEG    AT 0H           ; (6)
MAIN :  DW      START

        CSEG                    ; (7)
START :
                                ; chip initialize
        MOVW    AX , #_@STBEG
        MOVW    SP , AX

        MOV     HDTSA , #1AH
        MOVW    HL , #HDTSA     ; set hex 2-code data in HL register
        CALL    !CONVAH         ; convert ASCII <- HEX
                                ; output BC-register <- ASCII code

        MOVW    DE , #STASC     ; set DE <- store ASCII code table
        MOV     A , B
        MOV     [ DE ] , A
        INCW    DE
        MOV     A , C
        MOV     [ DE ] , A
        BR      $$

        END                     ; (8)
```

(1) Declaration of module name

(2) Declaration of symbol referenced from another module as an external reference symbol

(3) Declaration of symbol defined in another module as an external reference symbol

(4) Declaration of stack solution symbol generated from " -s " option of linker as an external reference symbol (an error will occur if " -s " option is not specified when linking)

(5) Declaration of the start of a data segment (to be located in saddr)

(6) Declaration of the start of a code segment (to be located as an absolute segment starting from address 0H)

(7) Declaration of the start of a code segment (meaning the end of the absolute segment)

(8) Declaration of the end of the module

< Subroutine >

```
        NAME    SAMPS           ; (1)
; ************************************************
;       HEX -> ASCII Conversion Program
;       sub-routine
;
;   input condition    : ( HL ) <- hex 2 code
;   output condition   : BC-register <- ASCII 2 code
; ************************************************

        PUBLIC  CONVAH          ; (2)

        CSEG                    ; (3)
CONVAH :
        MOV     A , [ HL ]
        ROR     A , 1
        ROR     A , 1
        ROR     A , 1
        ROR     A , 1
        AND     A , #0FH        ; hex upper code load
        CALL    !SASC
        MOV     B , A           ; store result

        XOR     A , A
        XCH     A , [ HL ]
        AND     A , #0FH        ; hex lower code load
        CALL    !SASC
        MOV     C , A           ; store result
        RET

 ; ********************************************
 ;       subroutine   convert ASCII code
 ;
 ;   input Acc ( lower 4bits )  <- hex code
 ;   output Acc                 <- ASCII code
 ; ********************************************

        CSEG
SASC :
        CMP     A , #0AH        ; check hex code > 9
        BC      $SASC1
        ADD     A , #07H        ; bias ( +7H )
SASC1 :
        ADD     A , #30H        ; bias ( +30H )
        RET

        END                     ; (4)
```

(1) Declaration of module name

(2) Declaration of symbol referenced from another module as an external definition symbol

(3) Declaration of the start of the code segment

(4) Declaration of the end of the module

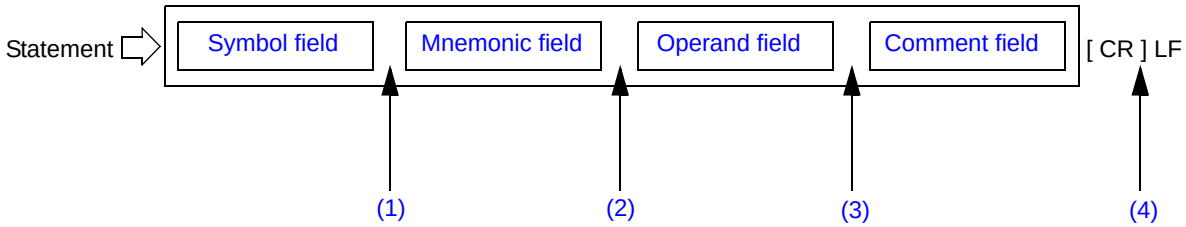
2.2 Description Method

2.2.1 Configuration

A source program consists of statements.

Each statement consists of the four fields shown in Figure 2-5.

Figure 2-5 Fields That Make Up a Statement



(1) The symbol field and the mnemonic field must be separated from each other with a colon (:) or one or more blanks or tabs (Whether colons or blanks are used depends on an instruction described in the mnemonic field).

(2) The mnemonic field and the operand field must be separated from each other with one or more blanks or tabs. Depending on the instruction described in the mnemonic field, the operand field may not be required.

(3) The comment field if used must be preceded with a semicolon (;).

(4) Each line must be delimited with an LF code (one CR code may exist immediately before the LF code).

A statement must be described within a line. A maximum of 2,048 characters (including CR and LF) can be described per line.

Each TAB or independent CR is counted as a single character. If 2,049 or more characters are described, a warning message is output and any characters at or over 2,049 are ignored. However, 2,049 or more characters will be output to the assembly list.

An independent CR will not be output to the assembly list.

The following lines may also be described :

- Dummy line (line without statement description)
- Line consisting of the symbol field alone
- Line consisting of the comment field alone

2.2.2 Character set

Characters that can be described in a source file are classified into the following three types :

- Language characters
- Character data
- Comment characters

(1) Language characters

Language characters are characters used to describe instructions in a source program. The language character set includes alphabetic, numeric, and special characters.

Table 2-2 Alphanumeric Characters

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

Table 2-3 Special Characters

Character	Name	Main Use	
? @ _	Question mark Circa Underscore	Symbol equivalent to alphabetic characters Symbol equivalent to alphabetic characters Symbol equivalent to alphabetic characters	
Blank HT (09H) , : ; CR (0DH) LF (0AH)	 Tab code Comma Colon Semicolon Carriage return code Line-feed code	Delimiter symbols	Delimiter of each field Character equivalent to blank Delimiter of operands Delimiter of labels Symbol indicating the start of the Comment field Symbol indicating the end of a line (ignored in the assembler) Symbol indicating the end of a line
+ - * / . (,) < , > =	Plus sign Minus sign Asterisk Slash Period Left and right parentheses Not Equal sign Equal sign	Assembler operators	ADD operator or positive sign SUBTRACT operator or negative sign MULTIPLY operator DIVIDE operator Bit position specifier Symbols specifying the order of arithmetic operations to be performed Relational operators Relational operator
'	Single quotation mark	- Symbol indicating the start or end of a character constant - Symbol indicating a complete macro parameter	

Table 2-3 Special Characters

Character	Name	Main Use
$	Dollar sign	- Symbol indicating the location counter - Symbol indicating the start of a control instruction equivalent to an assembler option - Symbol specifying relative addressing
&	Ampersand	Concatenating symbol (used in macro body)
#	Sharp sign	Symbol specifying immediate addressing
!	Exclamation point	Symbol specifying absolute addressing
[ ]	Brackets	Symbol specifying indirect addressing

(2) Character data

"Character data" refers to characters used to describe string constants, character strings, and control instructions (TITLE, SUBTITLE, INCLUDE).

Remark 1 All characters except "00H" can be used (including kanji (Japanese characters) ; codes may be different depending on the operating system). If "00H" has been described, an error will result and subsequent characters before the closing single quotation mark (') will be ignored.

Remark 2 If any illegal character has been described, the assembler will replace the illegal character with "!" for output to the assembly list (an independent CR (0DH) code will not be output to the assembly list).

Remark 3 With Windows, the assembler interprets code "1AH" as the end of the file (EOF) and thus the code cannot be a part of the input data.

(3) Comment characters

"Comment characters" refers to characters used to describe a comment statement.

Remark Characters that can be used in a comment statement are the same as those in the character set for character data. However, no error will result even if code "00H" has been described. Instead, the assembler will output the illegal character to the assembly list by replacing it with "!".

2.2.3 Symbol field

A symbol is described in the symbol field. The term "symbol" refers to a name given to numerical data or an address.

By using symbols, the contents of a source program can be understood more easily.

(1) Symbol Types

Symbols are classified into the types shown in Table 2-4, depending on their use and method of definition.

Table 2-4 Symbol Types

Symbol Type	Use	Method of Definition
Name	Used as numerical data or an address in a source program.	This type is described in the symbol field of the EQU, SET, or DBIT directive.
Label	Used as address data in a source program.	This type is defined by suffixing a colon (:) to a symbol.
External reference name	Used to reference symbol defined by a module by another module.	This type is described in the operand field of the EXTRN or EXTBIT directive.
Segment name	Symbol used during linker operation	This type is defined in the symbol field of the CSEG, DSEG, BSEG or ORG directive.
Module name	Used during symbolic debugging	This type is described in the operand field of the NAME directive.
Macro name	Used for macro reference in a source program.	This type is described in the symbol field of the MACRO directive.

(2) Conventions of Symbol Description

All symbols must be described according to the following rules :

(a) A symbol must be made up of alphanumeric characters and special characters (?, @, and _) that can be used as characters equivalent to alphabetic characters.

None of the numeric characters 0 to 9 can be used as the first character of a symbol.

(b) A symbol must be made up of not more than 255 characters. Characters in excess of the maximum symbol length will be ignored.

(c) No reserved word can be used as a symbol. Reserved words are indicated in Table A-2.

(d) The same symbol cannot be defined more than once (however, a name defined with the SET directive can be redefined with the SET directive).

(e) The assembler distinguishes between lowercase and uppercase characters.

(f) When describing a label in the Symbol field, ":" (colon) must be described immediately after the label.

< Examples of correct symbol descriptions >

```
CODE01  CSEG            ; "CODE01" is a segment name.
VAR01   EQU     10H     ; "VAR01" is a name.
LAB01 : DW      0       ; "LAB01" is a label.
        NAME    SAMPLE  ; "SAMPLE" is a module name.
MAC1    MACRO           ; "MAC1" is a macro name.
```

< Examples of incorrect symbol descriptions >

```
1ABC    EQU     3       ; No numeric character can be used as the 1st
                        ; character of a symbol.
LAB     MOV     A , R0  ; "LAB" is a label and must be separated from
                        ; the Mnemonic field with a colon ( : ).
FLAG :  EQU     10H     ; A colon ( : ) is not necessary in a name.
```

< Example of a symbol that is too long >

```
A123456789B12 to Y1234567890123456      EQU     70H
                  256   ; Character "6" in excess of the maximum symbol
                        ; length ( 255 characters ) are ignored.
                        ; The symbol will be defined as
                        ; "A123456789B12 to Y123456789012345".
```

< Example of a statement composed of a symbol only >

```
ABCD :                  ; "ABCD" will be defined as a label.
```

(3) Some Cautions about Symbols

The symbol "??RAnnnn (n = 0000 to FFFF)" is a symbol that is automatically replaced by the assembler every time a local symbol is developed inside a macro body. Be careful not to define this symbol twice.

When a segment name is not specified by a segment definition directive, the assembler generates a segment name automatically. These segments are shown in Table 2-5.

Duplicate segment name definition causes an error.

Table 2-5 Names of Segments Automatically Generated by Assembler

Segment Name	Directive	Relocation Attribute
?An (n = 0000 to FFFF)	ORG directive	(none)
?CSEG	CSEG directive	UNIT
?CSEGUP		UNITP
?CSEGT0		CALLT0
?CSEGFX		FIXED
?CSEGIX		IXRAM
?DSEG	DSEG directive	UNIT
?DSEGUP		UNITP
?DSEGS		SADDR
?DSEGSP		SADDRP
?DSEGIH		IHRAM
?DSEGL		LRAM
?DSEGDSP		DSPRAM
?DSEGIX		IXRAM
?BSEG	BSEG directive	UNIT

(4) Symbol Attributes

All names and labels have both a value and an attribute.

A value refers to the value of defined numerical data or address data itself.

Segment names, module names, and macro names do not have a value.

The attribute of a symbol is called a symbol attribute and must be one of the eight types indicated in Table 2-6.

Table 2-6 Symbol Attributes and Values

Attribute Type	Classification	Value
NUMBER	- Names to which numeric constants are assigned - Symbols defined with the EXTRN directive - Numeric constants	Decimal representation : 0 to 65535 Hexadecimal representation : 0H to FFFFH
ADDRESS	- Symbols defined as labels - Names defined as labels with EQU and SET directives	Decimal representation : 0 to 1048575 Hexadecimal representation : 0H to FFFFH
BIT	- Names defined as bit values - Names within BSEG - Symbols defined with the EXTBIT directive	saddr area
CSEG	Segment names defined with the CSEG directive	These attribute types have no value.
DSEG	Segment names defined with the DSEG directive	
BSEG	Segment names defined with the BSEG directive	
MODULE	Module names defined with the NAME directive (A module name if not defined is created from the primary name of the input source filename)	
MACRO	Macro names defined with the MACRO directive	

< Examples >

```
TEN     EQU     10H         ; Name "TEN" has attribute "NUMBER"
                            ; and value "10H".
        ORG     80H
START : MOV     A , #10H    ; Label "START" has attribute "ADDRESS"
                            ; and value "80H".
BIT1    EQU     0FE20H.0    ; Name "BIT1" has attribute "BIT"
                            ; and value "0FE20H.0".
```

2.2.4 Mnemonic field

In the mnemonic field, a mnemonic instruction, a directive, or a macro reference is described.

With an instruction or directive requiring an operand or operands, the mnemonic field must be separated from the operand field with one or more blanks or tabs.

However, with the first operand of an instruction that begins with "#", "$","!", or "[", the assembly will be executed properly even if nothing exists between the mnemonic field and the first operand field.

< Examples of correct descriptions >

```
MOV     A , #0H
CALL    !CONVAH
RET
```

< Examples of incorrect descriptions >

```
MOVA    #0H         ; No blank exists between the mnemonic and operand fields.
CALL    !CONVAH     ; A blank exists within the mnemonic field.
ZZZ                 ; The 78K0S Series has no such instruction as "ZZZ".
```

2.2.5 Operand field

In the operand field, the data (operands) required for executing the instruction, directive, or macro reference is described.

Depending on the instruction or directive, no operand is required in the operand field or two or more operands must be described in the operand field.

When describing two or more operands, delimit each operand with a comma (,).

The following types of data can be described in the operand field :

- Constants (numeric constants and string constants)
- Character Strings
- Register Names
- Special Characters ($, #, !, and [ ])
- Relocation attributes of segment definition directives
- Symbols
- Expressions
- Bit terms

The size and attribute of the required operand may be different depending on the instruction or directive. Refer to "2.6 Characteristics of Operands" for the sizes and attributes of operands.

For the operand representation formats and description methods in the instruction set, see the user's manual of the microcontroller for which software is being developed.

Each of the data types that can be described in the operand field is detailed below.

(1) Constants

A constant is a fixed value or data item and is also referred to as immediate data.

Constants are divided into numeric constants and character-string constants.

(a) Numeric constants

A binary, octal, decimal, or hexadecimal number can be described as a numeric constant.

The method of representing each numeric constant type is shown in Table 2-7 below.

A numeric constant will be processed as unsigned 16-bit data.

Value range : 0 ≤ n ≤ 65,535 (0FFFFH)

When describing a negative value, use the minus sign of the operator.

Table 2-7 Methods of Representing Numeric Constants

Constant	Method of Representation	Example
Binary constant	Character "B" or "Y" is suffixed to a numerical value.	1101B 1101Y
Octal constant	Character "O" or "Q" is suffixed to a numerical value.	74O 74Q
Decimal constant	A numerical value is described as is, or character "D" or "T" is suffixed to a numerical value.	128 128D 128T
Hexadecimal constant	- Character "H" is suffixed to a numerical value. - If the first character begins with "A", "B", "C", "D", "E", or "F", "0" must be prefixed to the constant.	8CH 0A6H

(b) Character-string constants

A character-string constant is expressed by enclosing a string of characters from those shown in "2.2.2 Character set", in a pair of single quotation marks (').

As a result of an assembly process, the character-string constant is converted into 7-bit ASCII code with the parity bit (MSB) set as "0".

The length of a string constant is 0 to 2 characters.

To use the single quotation mark itself as a string constant, the single quotation mark must be input twice in succession.

< Examples of character-string constant descriptions >

```
' ab '          ; Represents "6162H"
' A '           ; Represents "0041H"
' A '''         ; Represents "4127H"
' '             ; Represents "0020H" ( one blank )
```

(2) Character Strings

A character string is expressed by enclosing a string of characters from those shown in "2.2.2 Character set", in a pair of single quotation marks ('). Character strings are mainly used for operands in the DB directive and TITLE or SUBTITLE control instruction.

< Application examples of character strings >

```
        CSEG
MAS1 :  DB      ' YES ' ; Initializes with character string "YES".
MAS2 :  DB      ' NO '  ; Initializes with character string "NO".
```

(3) Register Names

The following registers can be described in the Operand field :

- General registers
- General register pairs
- Special function registers

General registers and general register pairs can be described with their absolute names (R0 to R7 and RP0 to RP3), as well as with their function names (X, A, B, C, D, E, H, L, AX, BC, DE, HL).

The register names that can be described in the operand field may differ depending on the type of instruction. For details of the method of describing each register name, see the user's manual of each device for which software is being developed.

(4) Special Characters

Special characters that can be described in the operand field are shown in Table 2-8.

Table 2-8 Special Characters That Can Be Described in Operand Field

Special Character	Function
$	- Indicates the location address of the instruction having this operand (or the 1st byte of this address, in the case of addresses with a multiple-byte instruction). - Indicates a relative addressing mode for a branch instruction.
!	- Indicates an absolute addressing mode for a branch instruction. - Indicates the specification of addr16 that allows all memory space to be specified with an MOV instruction.
#	- Indicates immediate data.
[]	- Indicates indirect addressing mode.

< Application examples of special characters >

```
Address Source program
100             ADD     A , #10H
102     LOOP :  INC     A
103             BR      $$ - 1          ; (1)
105             BR      !$ + 100H       ; (2)
```

(1) The second $ in the operand indicates address 103H. Describing "BR $ - 1" results in the same operation.

(2) The second $ in the operand indicates address 105H. Describing "BR $ + 100H" results in the same operation.

(5) Relocation attributes of segment definition directives

Relocation attributes can be described in the operand field.

For details of relocation attributes, refer to "3.2 Segment Definition Directives".

(6) Symbols

If a symbol is described in the operand field, an address (or value) allocated to that symbol becomes the operand value.

< Application examples of symbols >

```
VALUE   EQU     100H
        MOV     A , #VALUE      ; This description can be written as
                                ; "MOV A , #100H".
```

(7) Expressions

An expression is constants, $ (which indicates a location address), names, or labels connected with operators.

The expression can be described where numeric values can be expressed as instruction operands.

For the expressions and operators, refer to "2.3 Expressions and Operators".

< Examples of expressions >

```
TEN     EQU     10H
        MOV     A , #TEN - 5H
```

In this example, "TEN - 5H" is an expression.

In this expression, the name and numeric constant are connected with a - (minus) operator. The value of the expression is BH.

Therefore, this description can be rewritten as "MOV A , #0BH".

(8) Bit terms

A bit term can be obtained by the bit position specifier. For details of bit terms, refer to 2.5 Bit Position Specifier.

< Examples of bit terms >

```
CLR1    A.5
SET1    1 + 0FE30H.3            ; The operand value is 0FE31H.3.
CLR1    0FE40H.4 + 2            ; The operand value is 0FE40H.6.
```

2.2.6 Comment field

In the comment field, comments or remarks may be described following the input of a semicolon (;). The comment field is from a semicolon to the line-feed code of that line or EOF. By describing a comment statement in the comment field, an easy-to-understand source program can be created. The comment statement in the comment field is not subject to assembler operation (i.e., conversion into machine language) but will be output without change on an assembly list.

Characters that can be described in the comment field are those shown in "2.2.2 Character set".

< Examples of comments >

```
        NAME    SAMPM
; ************************************
;       HEX -> ASCII Conversion Program
;       main-routine
; ************************************

        PUBLIC  MAIN , START
        EXTRN   CONVAH
        EXTRN   @STBEG

DATA    DSEG    saddr
HDTSA:  DS      1
STASC:  DS      2

CODE    CSEG    AT 0H
MAIN :  DW      START

        CSEG
START :
                                ; chip initialize
        MOVW    AX , #_@STBEG
        MOVW    SP , AX

        MOV     HDTSA , #1AH
        MOVW    HL , #HDTSA     ; set hex 2-code data in HL register

        CALL    !CONVAH         ; convert ASCII <- HEX
                                ; output BC-register <- ASCII code

        MOVW    DE , #STASC     ; set DE <- store ASCII code table
        MOV     A , B
        MOV     [ DE ] , A
        INCW    DE
        MOV     A , C
        MOV     [ DE ] , A
        BR      $$

        END
```

Lines consisting of comment field only

Lines consisting of comment field only

Lines in which comments are described in comment field

2.3 Expressions and Operators

An expression is a symbol, constant, location address (indicated by $) or bit term, an operator combined with one of the above, or a combination of operators.

Elements of an expression other than the operators are called terms, and are referred to as the 1st term, 2nd term, and so forth from left to right, in the order of their description.

Operators are available in the types shown in Table 2-9, and the order of their precedence in calculation has been predetermined as shown in Table 2-10.

Parentheses "( )" are used to change the order in which calculations are performed.

< Example >

```
MOV     A , #5 * ( SYM + 1 )    ; ( 1 )
```

In (1) above, "5 * (SYM + 1)" is an expression. "5" is the 1st term of the expression and "SYM" and "1" are the 2nd and 3rd terms respectively. "*", "+", and "( )" are operators.

Table 2-9 Types of Operators

Type of Operator	Operators
Arithmetic Operators	+, -, *, /, MOD, + sign, - sign
Logical Operators	NOT, AND, OR, XOR
Relational Operators	EQ (or =), NE (or < >), GT (or >), GE (or >=), LT (or <), LE (or <=)
Shift Operators	SHR, SHL
Byte-Separating Operators	HIGH, LOW
Special Operators	DATAPOS, BITPOS, MASK
Other Operators	( )

The above operators can also be divided into unary operators, special unary operators, binary operators, N-ary operators, and other operators.

Unary operators	+ sign, - sign, NOT, HIGH, LOW
Special unary operators	DATAPOS, BITPOS
Binary operators	+, -, *, /, MOD, AND, OR, XOR, EQ (or =), NE (or < >), GT (or >), GE (or >=), LT (or <), LE (or <=), SHR, SHL
N-ary operators	MASK
Other operators	( )

Table 2-10 Order of Precedence of Operators

Priority	Priority Level	Operators
Higher	1	+ sign, - sign, NOT, HIGH, LOW, DATAPOS, BITPOS, MASK
	2	*, /, MOD, SHR, SHL
	3	+, -
	4	AND
	5	OR, XOR
Lower	6	EQ (or =), NE (or < >), GT (or >), GE (or >=), LT (or <), LE (or <=)

Operations on expressions are performed according to the following rules :

(1) Operations are performed according to the order of precedence given to each operator. If two or more operators of the same order of precedence exist in an expression, the operation designated by the leftmost operator will be carried out. In the case of unary operators, the operation will be performed from right to left.

(2) An expression in parentheses is carried out before expressions outside the parentheses.

(3) Operations between two or more unary operators are allowed.

Examples : 1 = --1 == 1

-1 = -+1 = -1

(4) Expressions are calculated within 16 bits, without signs. If an overflow occurs in operation due to an expression exceeding 16 bits, the overflowed value is ignored.

(5) If a constant exceeds 16 bits (0FFFFH), an error will result and the value of the result will be regarded as 0 for calculation.

(6) In division, the decimal fraction part of the result will be truncated. If the divisor is 0, an error will occur, and the result will be 0.

2.3.1 Operators

The functions of the respective operators are described in this subsection.

Arithmetic Operators

(1) +

[Function]

- Returns the sum of the values of the 1st and 2nd terms of an expression.

[Application Example]

```
        ORG     100H
START : BR      !$ + 6          ; ( a )
```

[Explanation]

- The BR instruction causes a jump to "current location address plus 6", namely, to address "100H + 6H = 106H".

  Therefore, (a) in the above example can also be described as : START : BR !106H

(2) -

[Function]

- Returns the result of subtraction of the 2nd-term value from the 1st-term value.

[Application Example]

```
        ORG     100H
BACK :  BR      BACK - 6H       ; ( b )
```

[Explanation]

- The BR instruction causes a jump to "address assigned to BACK minus 6", namely, to address "100H - 6H = 0FAH".

  Therefore, (b) in the above example can also be described as : BACK : BR !0FAH

(3) *

[Function]

- Returns the result of multiplication (product) between the values of the 1st and 2nd terms of an expression.

[Application Example]

```
TEN     EQU     10H
        MOV     A , #TEN * 3    ; ( c )
```

[Explanation]

- With the EQU directive, the value "10H" is defined in the name "TEN".

  "#" indicates immediate data. The expression "TEN * 3" is the same as "10H * 3" and returns the value "30H".

  Therefore, (c) in the above expression can also be described as : MOV A , #30H

(4) /

[Function]

- Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result. The decimal fraction part of the result will be truncated. If the divisor (2nd term) of a division operation is 0, an error will result.

[Application Example]

```
MOV     A , #256 / 50           ; ( d )
```

[Explanation]

- The result of the division "256 / 50" is 5 with remainder 6.
  The operator returns the value "5" that is the integer part of the result of the division.
  Therefore, (d) in the above expression can also be described as : MOV A , #5

(5) MOD

[Function]

- Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term.
  An error will result if the divisor (2nd term) is 0.
  A blank is required before and after the MOD operator.

[Application Example]

```
MOV     A , #256 MOD 50         ; ( e )
```

[Explanation]

- The result of the division "256 / 50" is 5 with remainder 6.
  The MOD operator returns the remainder 6.
  Therefore, (e) in the above expression can also be described as : MOV A , #6.

(6) + sign

[Function]

- Returns the value of the term of an expression without change.

[Application Example]

```
FIVE    EQU     +5
```

[Explanation]

- The value "5" of the term is returned without change.
  The value "5" is defined in name "FIVE" with the EQU directive.

(7) - sign

[Function]

- Returns the value of the term of an expression by the two's complement.

[Application Example]

```
NO      EQU     -1
```

[Explanation]

- -1 becomes the two's complement of 1.

  The two's complement of binary 0000 0000 0000 0001 becomes :

  1111 1111 1111 1111

  Therefore, with the EQU directive, the value "0FFFFH" is defined in the name "NO".

Logical Operators

(1) NOT

[Function]

- Negates the value of the term of an expression on a bit-by-bit basis and returns the result.
  A blank is required between the NOT operator and the term.

[Application Example]

```
MOVW    AX , #NOT 3H            ; ( a )
```

[Explanation]

- Logical negation is performed on "3H" as follows :

NOT)	0000	0000	0000	0011
	1111	1111	1111	1100

0FFFCH is returned.
Therefore, (a) can also be described as : MOVW AX , #0FFFCH

(2) AND

[Function]

- Performs an AND (logical product) operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.
  A blank is required before and after the AND operator.

[Application Example]

```
MOV     A , #6FAH AND 0FH       ; ( b )
```

[Explanation]

- AND operation is performed between the two values "6FAH" and "0FH" as follows :

	0000	0110	1111	1010
AND)	0000	0000	0000	1111
	0000	0000	0000	1010

The result "0AH" is returned. Therefore, (b) in the above expression can also be described as : MOV A , #0AH

(3) OR

[Function]

- Performs an OR (Logical sum) operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result.
  A blank is required before and after the OR operator.

[Application Example]

```
MOV     A , #0AH OR 1101B       ; ( c )
```

[Explanation]

- OR operation is performed between the two values "0AH" and "1101B" as follows :

	0000	0000	0000	1010
OR)	0000	0000	0000	1101
	0000	0000	0000	1111

The result "0FH" is returned.
Therefore, (c) in the above expression can also be described as : MOV A , #0FH

(4) XOR

[Function]

- Performs an Exclusive-OR operation between the value of the 1st term of an expression and the value of its 2nd term on a bit-by-bit basis and returns the result. A blank is required before and after the XOR operator.

[Application Example]

```
MOV     A , #9AH XOR 9DH        ; ( d )
```

[Explanation]

- XOR operation is performed between the two values "9AH" and "9DH" as follows :

	0000	0000	1001	1010
XOR)	0000	0000	1001	1101
	0000	0000	0000	0111

The result "7H" is returned.
Therefore, (d) in the above expression can also be described as : MOV A , #7H

Relational Operators

(1) EQ (or =)

[Function]

- Returns 0FFH (True) if the value of the 1st term of an expression is equal to the value of its 2nd term, and 00H (False) if both values are not equal.

  A blank is required before and after the EQ operator.

[Application Example]

```
A1      EQU     12C4H
A2      EQU     12C0H

        MOV     A , #A1 EQ ( A2 + 4H )  ; ( a )
        MOV     X , #A1 EQ A2           ; ( b )
```

[Explanation]

- In (a) above, the expression "A1 EQ (A2 + 4H)" becomes "12C4H EQ (12C0H + 4H)".

  The operator returns 0FFH because the value of the 1st term is equal to the value of the 2nd term.
- In (b) above, the expression "A1 EQ A2" becomes "12C4H EQ 12C0H".

  The operator returns 00H because the value of the 1st term is not equal to the value of the 2nd term.

(2) NE (or < >)

[Function]

- Returns 0FFH (True) if the value of the 1st term of an expression is not equal to the value of its 2nd term, and 00H (False) if both values are equal.

  A blank is required before and after the NE operator.

[Application Example]

```
A1      EQU     5678H
A2      EQU     5670H

        MOV     A , #A1 NE A2           ; ( c )
        MOV     A , #A1 NE ( A2 + 8H )  ; ( d )
```

[Explanation]

- In (c) above, the expression "A1 NE A2" becomes "5678H NE 5670H".

  The operator returns 0FFH because the value of the 1st term is not equal to the value of the 2nd term.
- In (d) above, the expression "A1 NE (A2 + 8H)" becomes "5678H NE (5670H + 8H)".

  The operator returns 00H because the value of the 1st term is equal to the value of the 2nd term.

(3) GT (or >)

[Function]

- Returns 0FFH (True) if the value of the 1st term of an expression is greater than the value of its 2nd term, and 00H (False) if the value of the 1st term is equal to or less than the value of the 2nd term.

  A blank is required before and after the GT operator.

[Application Example]

```
A1      EQU     1023H
A2      EQU     1013H

        MOV     A , #A1 GT A2           ; ( e )
        MOV     X , #A1 GT ( A2 + 10H ) ; ( f )
```

[Explanation]

- In (e) above, the expression "A1 GT A2" becomes "1023H GT 1013H".

  The operator returns 0FFH because the value of the 1st term is greater than the value of the 2nd term.

- In (f) above, the expression "A1 GT (A2 + 10H)" becomes "1023H GT (1013H + 10H)".

  The operator returns 00H because the value of the 1st term is equal to the value of the 2nd term.

(4) GE (or >=)

[Function]

- Returns 0FFH (True) if the value of the 1st term of an expression is greater than or equal to the value of its 2nd term, and 00H (False) if the value of the 1st term is less than the value of the 2nd term.
- A blank is required before and after the GE operator.

[Application Example]

```
A1      EQU     2037H
A2      EQU     2015H

        MOV     A , #A1 GE A2           ; ( g )
        MOV     X , #A1 GE ( A2 + 23H ) ; ( h )
```

[Explanation]

- In (g) above, the expression "A1 GE A2" becomes "2037H GE 2015H".

  The operator returns 0FFH because the value of the 1st term is greater than the value of the 2nd term.

- In (h) above, the expression "A1 GE (A2 + 23H)" becomes "2037H GE (2015H + 23H)".

  The operator returns 00H because the value of the 1st term is less than the value of the 2nd term.

(5) LT (or <)

[Function]

- Returns 0FFH (True) if the value of the 1st term of an expression is less than the value of its 2nd term, and 00H (False) if the value of the 1st term is equal to or greater than the value of the 2nd term.

  A blank is required before and after the LT operator.

[Application Example]

```
A1      EQU     1000H
A2      EQU     1020H

        MOV     A , #A1 LT A2                   ; ( i )
        MOV     X , # ( A1 + 20H ) LT A2        ; ( j )
```

[Explanation]

- In (i) above, the expression "A1 LT A2" becomes "1000H LT 1020H".

  The operator returns 0FFH because the value of the 1st term is less than the value of the 2nd term.

- In (j) above, the expression "(A1 + 20H) LT A2" becomes "(1000H + 20H) LT 1020H".

  The operator returns 00H because the value of the 1st term is equal to the value of the 2nd term.

(6) LE (or <=)

[Function]

- Returns 0FFH (True) if the value of the 1st term of an expression is less than or equal to the value of its 2nd term, and 00H (False) if the value of the 1st term is greater than the value of the 2nd term.

  A blank is required before and after the LE operator.

[Application Example]

```
A1      EQU     103AH
A2      EQU     1040H

        MOV     A , #A1 LE A2                   ; ( k )
        MOV     X , # ( A1 + 7H ) LE A2         ; ( l )
```

[Explanation]

- In (k) above, the expression "A1 LE A2" becomes "103AH LE 1040H".

  The operator returns 0FFH because the value of the 1st term is less than the value of the 2nd term.

- In (l) above, the expression "(A1 + 7H) LE A2" becomes "(103AH + 7H) LE 1040H".

  The operator returns 00H because the value of the 1st term is greater than the value of the 2nd term.

Shift Operators

(1) SHR

[Function]

- Returns a value obtained by shifting the value of the 1st term of an expression to the right the number of bits specified by the value of the 2nd term. Zeros equivalent to the specified number of bits shifted move into the high-order bits.

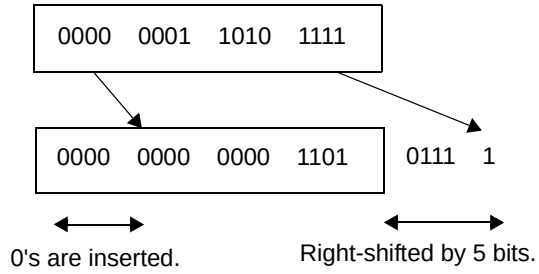
  A blank is required before and after the SHR operator.

[Application Example]

```
MOV     A , #01AFH SHR 5        ; ( a )
```

[Explanation]

- This operator shifts the value "01AFH" to the right by 5 bits.



The value "000DH" is returned.

Therefore, (a) in the above example can also be described as : MOV A , #0DH

(2) SHL

[Function]

- Returns a value obtained by shifting the value of the 1st term of an expression to the left the number of bits specified by the value of the 2nd term. Zeros equivalent to the specified number of bits shifted move into the low-order bits.

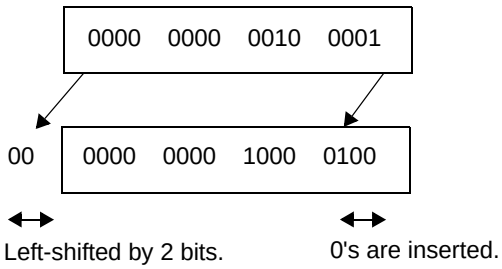
  A blank is required before and after the SHL operator.

[Application Example]

```
MOV     A , #21H SHL 2          ; ( b )
```

[Explanation]

- This operator shifts the value "21H" to the left by 2 bits.



The value "84H" is returned.

Therefore, (b) in the above example can also be described as : MOV A , #84H

Byte-Separating Operators

(1) HIGH

[Function]

- Returns the high-order 8-bit value of a term.

  A blank is required between the HIGH operator and the term.

[Application Example]

```
MOV     A , #HIGH 1234H          ; ( a )
```

[Explanation]

- By executing a MOV instruction, this operator returns the high-order 8-bit value "12H" of the expression "1234H".

  Therefore, (a) in the above example can also be described as : MOV A , #12H

(2) LOW

[Function]

- Returns the low-order 8-bit value of a term.

  A blank is required between the LOW operator and the term.

[Application Example]

```
MOV     A , #LOW 1234H           ; ( b )
```

[Explanation]

- By executing a MOV instruction, this operator returns the low-order 8-bit value "34H" of the expression "1234H".

  Therefore, (b) in the above example can also be described as : MOV A , #34H

Special Operators

(1) DATAPOS

[Function]

- Returns the address portion (byte address) of a bit symbol.

[Application Example]

```
SYM     EQU     0FE68H.6

        MOV     A , !DATAPOS SYM        ; ( a )
```

[Explanation]

- An EQU directive defines the name "SYM" with a value of 0FE68H.6.
  "DATAPOS SYM" represents "DATAPOS 0FE68H.6", and "0FE68H" is returned.
  Therefore, (a) in the above example can also be described as : MOV A , !0FE68H

(2) BITPOS

[Function]

- Returns the bit portion (bit position) of a bit symbol.

[Application Example]

```
SYM     EQU     0FE68H.6

        CLR1    [ HL ] .BITPOS SYM      ; ( b )
```

[Explanation]

- An EQU directive defines the name "SYM" with a value of 0FE68H.6.
  "BITPOS.SYM" represents "BITPOS 0FE68H.6", and "6" is returned.
  A CLR1 instruction clears [HL].6 to 0.

(3) MASK

[Function]

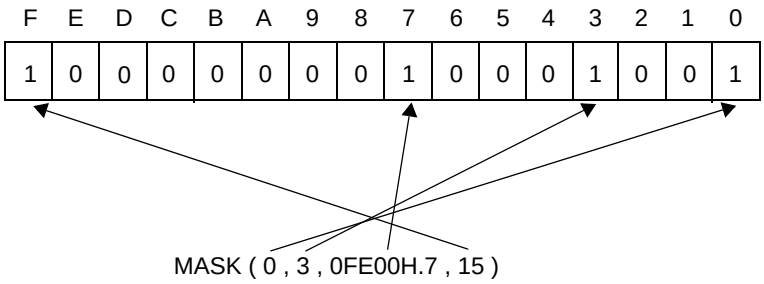
- Returns a 16-bit value in which the specified bit position is 1 and all others are set to 0.

[Application Example]

```
MOVW    AX , #MASK ( 0 , 3 , 0FE00H.7 , 15 )
```

[Explanation]

- A MOVW instruction returns the value "8089H".



Other Operators

(1) ( )

[Function]

- Causes an operation in parentheses to be performed prior to operations outside the parentheses.

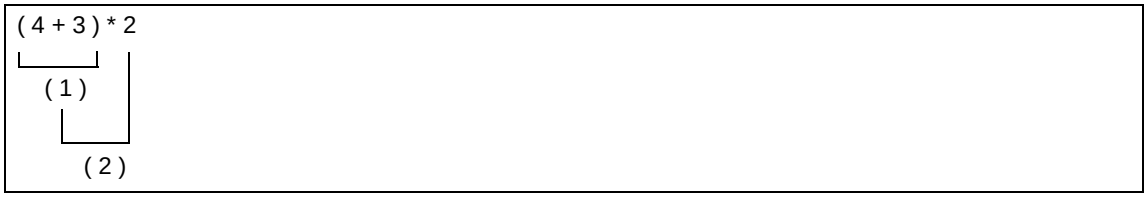
  This operator is used to change the order of precedence of other operators.

  If parentheses are nested at multiple levels, the expression in the innermost parentheses will be calculated first.

[Application Example]

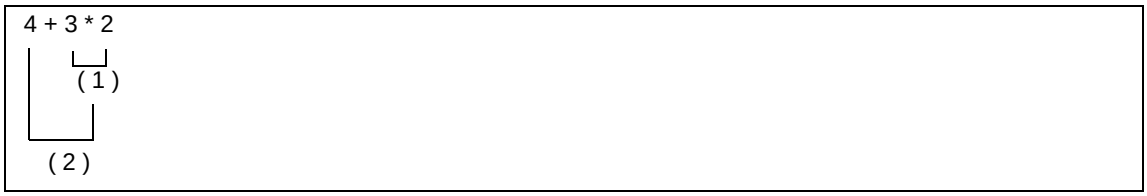
```
MOV     A , # ( 4 + 3 ) * 2
```

[Explanation]



Calculations are performed in the order of expressions (1), (2) and the value "14" is returned as a result.

If parentheses are not used,



Calculations are performed in the order (1), (2) shown above, and the value "10" is returned as a result.

See Table 2-10, for the order of precedence of operators.

2.4 Restrictions on Operations

The operation of an expression is performed by connecting terms with operator(s). Elements that can be described as terms include constants, $, names, and labels. Each term has a relocation attribute and a symbol attribute.

Depending on the types of relocation attribute and symbol attribute inherent in each term, operators that can work on the term are limited. Therefore, when describing an expression, it is important to pay attention to the relocation attribute and symbol attribute of each of the terms constituting the expression.

2.4.1 Operators and relocation attributes

As previously mentioned, each of the terms that constitute an expression has a relocation attribute and symbol attribute.

Terms can be divided into three types when classified by their relocation attributes : Absolute terms, relocatable terms, and external reference terms.

Types of relocation attributes in operations, the nature of each attribute, and terms applicable to each attribute are shown in Table 2-11.

Table 2-11 Types of Relocation Attributes

Type	Nature	Applicable Terms
Absolute term	Term whose value and constant are determined at assembly time	- Constants - Labels defined within an absolute segment - $ indicating the location address defined within an absolute segment - Names defined with constants, the above labels, the above $, or absolute values
Relocatable term	Term whose value is not determined at assembly time	- Labels defined within a relocatable segment - $ indicating the location address defined within a relocatable segment - Names defined with a relocatable symbol
External reference term[Note]	Term that externally references the symbol of another module	- Labels defined with the EXTRN directive - Names defined with the EXTBIT directive

Note The following four operators can work on external reference terms : "+", "-", "HIGH", and "LOW". Only one external reference symbol can be described in an expression. In this case, the external reference symbol must be connected with a "+" operator.

Combinations of the type of operator and terms on which each operator can work are shown in Table 2-12.

Table 2-12 Combinations of Terms and Operators by Relocation Attribute (Relocatable Terms)

Relocation Attribute of Term Type of Operator	X : ABS Y : ABS	X : ABS Y : REL	X : REL Y : ABS	X : REL Y : REL
X + Y	A	R	R	-
X - Y	A	-	R	A[Note 1]
X * Y	A	-	-	-
X / Y	A	-	-	-
X MOD Y	A	-	-	-
X SHL Y	A	-	-	-
X SHR Y	A	-	-	-
X EQ Y	A	-	-	A[Note 1]
X LT Y	A	-	-	A[Note 1]
X LE Y	A	-	-	A[Note 1]
X GT Y	A	-	-	A[Note 1]
X GE Y	A	-	-	A[Note 1]
X NE Y	A	-	-	A[Note 1]
X AND Y	A	-	-	-
X OR Y	A	-	-	-
X XOR Y	A	-	-	-
NOT X	A	A	-	-
+ X	A	A	R	R
- X	A	A	-	-
HIGH X	A	A	R[Note 2]	R[Note 2]
LOW X	A	A	R[Note 2]	R[Note 2]
MASK (X)	A	A	-	-
DATAPOS X.Y	A	-	-	-
BITPOS X.Y	A	-	-	-
MASK (X.Y)	A	-	-	-
DATAPOS X	A	A	R	R
BITPOS X	A	A	A	A

< Table Explanation >

ABS : Absolute term

REL : Relocatable term

A : The result of the operation becomes an absolute term.

R : The result of the operation becomes a relocatable term.

- : The operation cannot be performed.

Notes 1. The operation can only be performed if X and Y are defined within the same segment, and not relocatable terms on which HIGH, LOW, DATAPOS are operated.

Notes 2. The operation can only be performed if X and Y are not relocatable terms on which HIGH, LOW,DATAPOS are operated.

The following five operators can work on external reference terms : "+", "-", "HIGH", and "LOW" (however, note that only one external reference term can be described in an expression).

Combinations of the types of operators and external reference terms on which each operator can work areclassified according to relocation attributes in Table 2-13.

Table 2-13 Combinations of Terms and Operators by Relocation Attribute (External Reference Terms)

Relocation Attribute of Term Type of Operator	X : ABS Y : EXT	X : EXT Y : ABS	X : REL Y : EXT	X : EXT Y : REL	X : EXT Y : EXT
X + Y	E	E	-	-	-
X - Y	-	E	-	-	-
+ X	A	E	R	E	E
HIGH X	A	E Note 1	R Note 2	E Note 1	E Note 1
LOW X	A	E Note 1	R Note 2	E Note 1	E Note 1
MASK (X)	A	-	-	-	-
DATAPOS X.Y	-	-	-	-	-
BITPOS X.Y	-	-	-	-	-
MASK (X.Y)	-	-	-	-	-
DATAPOS X	A	E	R	E	E
BITPOS X	A	E	A	E	E

< Table Explanation >

ABS : Absolute term

EXT : External reference terms

REL : Relocatable term

A : The result of the operation becomes an absolute term.

E : The result of the operation becomes an external reference term.

R : The result of the operation becomes a relocatable term.

- : The operation cannot be performed.

Notes 1. The operation can only be performed if X and Y are not external reference terms on which HIGH, LOW, DATAPOS, BITPOS are operated.

Notes 2. The operation can only be performed if X and Y are not relocatable terms on which HIGH, LOW, DATAPOS are operated.

2.4.2 Operators and symbol attributes

As previously mentioned, each of the terms that constitute an expression has a symbol attribute in addition to a relocation attribute. Terms can be divided into two types when classified by their symbol attributes : NUMBER terms and ADDRESS terms.

Types of symbol attributes in operations and terms applicable to each attribute are shown in Table 2-14.

Table 2-14 Types of Symbol Attributes in Operations

Type of Symbol Attribute	Applicable Terms
NUMBER term	- Symbols that have NUMBER attribute - Constants
ADDRESS term	- Symbols that have ADDRESS attribute - $ indicating the location counter

Combinations of the type of operator and terms on which each operator can work when classified by their symbol attributes are shown in Table 2-15.

Table 2-15 Combinations of Terms and Operators by Symbol Attribute

Symbol Attribute of Term Type of Operator	X : ADDRESS Y : ADDRESS	X : ADDRESS Y : NUMBER	X : NUMBER Y : ADDRESS	X : NUMBER Y : NUMBER
X + Y	-	A	A	N
X - Y	N	A	-	N
X * Y	-	-	-	N
X / Y	-	-	-	N
X MOD Y	-	-	-	N
X SHL Y	-	-	-	N
X SHR Y	-	-	-	N
X EQ Y	N	-	-	N
X LT Y	N	-	-	N
X LE Y	N	-	-	N
X GT Y	N	-	-	N
X GE Y	N	-	-	N
X NE Y	N	-	-	N
X AND Y	-	-	-	N
X OR Y	-	-	-	N
X XOR Y	-	-	-	N
NOT X	-	-	N	N
+ X	A	A	N	N
- X	-	-	N	N
HIGH X	A	A	N	N
LOW X	A	A	N	N
DATAPOS X	A	A	N	N
MASK X	N	N	N	N

< Table Explanation >

ADDRESS : ADDRESS term

NUMBER : NUMBER term

A : The result of the operation becomes an ADDRESS term.

N : The result of the operation becomes a NUMBER term.

- : The operation cannot be performed.

2.4.3 How to check restrictions on the operation

An example of an operation by the relocation attribute and by symbol attribute of each term is shown here.

< Example >

```
BR      $TABLE + 5H
```

Here, assume that "TABLE" is a label defined in a relocatable code segment.

(a) Operator and relocation attribute

Because "TABLE + 5H" is "relocatable term+absolute term", this operation is applied to Table 2-12.

Type of operator : X + Y

Relocation attribute of term : X : REL, Y : ABS

From the table, you will find that the result is R (namely, a relocatable term).

(b) Operator and symbol attribute

Because "TABLE + 5H" is "ADDRESS term+NUMBER term", this operation is applied to Table 2-15.

Type of operator : X + Y

Symbol attribute of term : X : ADDRESS, Y : NUMBER

From the table, you will find that the result is A (namely, an ADDRESS term).

2.5 Bit Position Specifier

Bits can be accessed by using the bit position specifier (.).

Bit Position Specifier

(1) Period (.)

[Description Format]

```
  X [ Δ] . [ Δ] Y
└──────────┘
    Bit term
```

Table 2-16 Combinations of X (1st Term) and Y (2nd Term)

X (1st Term)		Y (2nd Term)
General register	A	Expression (0 to 7)
Control register	PSW	Expression (0 to 7)
Special function register	sfr[Note]	Expression (0 to 7)
Memory	[HL][Note]	Expression (0 to 7)

Note For details on the specific description, see the user's manual of each device.

[Function]

- The bit position specifier specifies a byte address with its 1st term and the position of a bit by its 2nd term. A specific bit can be accessed by this bit position specifier.

[Explanation]

- A bit term refers to an expression that uses a bit position specifier.
- The bit position specifier is not affected by the precedence order of operators. The left side of the bit position specifier is recognized as the 1st term and its right side as the 2nd term.
- The following restrictions apply to the 1st term :
  (1) An expression with the NUMBER or ADDRESS attribute, an SFR name capable of bit access or register name (A) can be described.
  (2) When an absolute expression is described in the 1st term, it must be within the range 0FE20H to 0FF1FH.
  (3) An external reference symbol can be described.
- The following restrictions apply to the 2nd term :
  (1) The value of an expression must be in the range of 0 to 7. If this value range is exceeded, an error will result.
  (2) Only an absolute expression with the NUMBER attribute can be described.
  (3) No external reference symbol can be described.

[Operations and Relocation Attributes]

- Combinations of the 1st and 2nd terms by relocation attribute are shown in Table 2-17.

Table 2-17 Combinations of 1st and 2nd Terms by Relocation Attribute

Combination of Terms X :	ABS	ABS	REL	REL	ABS	EXT	REL	EXT	EXT
Combination of Terms Y :	ABS	REL	ABS	REL	EXT	ABS	EXT	REL	EXT
X.Y	A	-	R	-	-	E	-	-	-

< Table Explanation >

ABS : Absolute term

EXT : External reference terms

REL : Relocatable term

A : The result of the operation becomes an absolute term.

E : The result of the operation becomes an external reference term.

R : The result of the operation becomes a relocatable term.

- : The operation cannot be performed.

[Values of Bit Symbols]

- When a bit symbol is defined by describing a bit term using the bit position specifier in the operand field of the EQU directive, the value that the bit symbol will have is shown in Table 2-18, below.

Table 2-18 Values of Bit Symbols

Operand Type	Symbol Value
A.bit[Note 2]	1.bit
PSW.bit[Note 2]	1FEH.bit
sfr[Note 1].bit[Note 2]	FFxxH.bit[Note 3]
expression.bit[Note 2]	xxxxH.bit[Note 4]

Notes 1. For a detailed description, refer to the user's manual of each device.

Notes 2. bit = 0 to 7

Notes 3. FFxxH indicates the address of an sfr.

Notes 4. xxxxH indicates the value of an expression.

[Application Example]

```
SET1    0FE20H.3
SET1    A.5
CLR1    P1.2
SET1    1 + 0FE30H.3    ; Equals 0FE31H.3
SET1    0FE40H.4 + 2    ; Equals 0FE40H.6
```

2.6 Characteristics of Operands

Instructions and directives requiring an operand or operands differ from one type of instruction to another in the size and address range of the required operand value and in the symbol attribute of the operand.

For example, the instruction "MOV r, #byte" functions to transfer the value indicated by "byte" to register "r". In this case, because r is an 8-bit register, the size of the data "byte" to be transferred must be 8 bits or less.

If an instruction is described as "MOV R0, #100H", an assembly error occurs, because the size of the 2nd operand "100H" of the instruction exceeds the capacity of the 8-bit register R0.

When describing an operand, therefore, attention must be paid to the following points :

- Is the size of the operand value or its address range suitable for the operand (numerical data, name, or label) of the instruction?
- Is the symbol attribute suitable for the operand (name or label) of the instruction?

2.6.1 Size and address range of operand value

Certain conditions are set for the size and address range of the value of the numerical data, name, or label that can be described as the operand of an instruction.

With instructions, conditions for the size and address range of an operand value are governed by the operand representation format of each instruction. With directives, conditions for the size and address range of an operand value are governed by the type of instructions.

These conditions are shown in Table 2-19 and Table 2-20, below.

Table 2-19 Ranges of Operand Values of Instructions

Operand Representation Format	Range of Values	
byte	8-bit value 0H to FFH	
word	16-bit value 0H to FFFFH	
saddr	FE20H to FF1FH	
saddrp	Even value of FE20H to FF1FH	
sfr	FF00H to FFCFH, FFE0H to FFFFH	
sfrp	Even value of FF00H to FFCFH, FFE0H to FFFFH	
addr16	MOV, MOVW	0H to FFFFH
	Other instructions	0H to FA7FH
addr5	Even value of 40H to 7EH	
bit	3-bit value 0 to 7	
n	2-bit value 0 to 3	

Table 2-20 Ranges of Operand Values of Directives

Type of Directive	Directive	Range of Values
Segment definition directives	CSEG AT	0H to FFFFH
	DSEG AT	0H to FFFFH
	BSEG AT	FE20H to FEFFH
	ORG	0H to FFFFH
Symbol definition directives	EQU	16-bit value 0H to FFFFH
	SET	16-bit value 0H to FFFFH
Memory initialization and area reservation directives	DB	8-bit value 0H to FFH
	DW	16-bit value 0H to FFFFH
	DS	16-bit value 0H to FFFFH
Automatic branch instruction selection directive	BR	0H to FFFFH

2.6.2 Size of operands required for instructions

Instructions can be classified into machine instructions and directives. For instructions that require immediate data and symbols as operands, the size of the operand required varies for each instruction.

Therefore, when data in excess of the size of the operand required for the instruction is described, an error occurs. The operations of expressions are carried out with unsigned 16 bits. If the evaluation result exceeds 0FFFFH (16 bits), a warning message is output.

However, when relocatable or external-reference symbols are described in an operand, the values are not determined within the assembler. Instead, the linker determines the values and checks the value range.

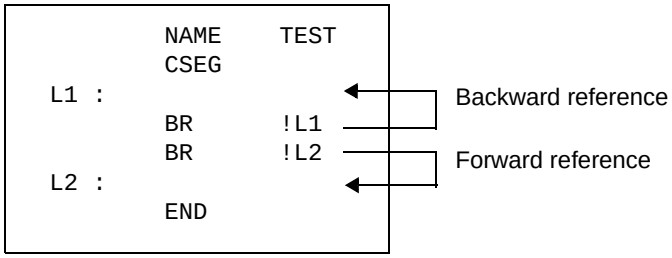
2.6.3 Symbol attributes and relocation attributes of operands

When names, labels, and $ (which indicate location counters) are described as instruction operands, they may or may not be describable as operands. This depends on the symbol attributes and relocation attributes (see "2.4 Restrictions on Operations") that serve as the terms of their expressions, as well as on the direction of reference in the case of names and labels.

Reference direction for names and labels can be backward reference or forward reference.

- Backward reference : A name or label referenced as an operand, which is defined in a line above (before) the name or label
- Forward reference : A name or label referenced as an operand, which is defined in a line below (after) the name or label

< Example >



These symbol attributes and relocation attributes, as well as direction of reference for names and labels, are shown in Table 2-21, and Table 2-22.

Table 2-21 Properties of Described Symbols as Operands

Symbol Attributes	NUMBER		ADDRESS				NUMBER ADDRESS		sfr Reserved Words[Note 1]
Relocation Attributes	Absolute Terms		Absolute Terms		Relocatable Terms		External Reference Terms		
Reference Pattern / Description Format	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	
byte	OK	OK	OK	OK	OK	OK	OK	OK	NG
word	OK	OK	OK	OK	OK	OK	OK	OK	NG
saddr	OK	OK	OK	OK	OK	OK	OK	OK	OK[Notes 2, 3]
saddrp	OK	OK	OK	OK	OK	OK	OK	OK	OK[Notes 2, 4]
sfr	OK Note 5	NG	NG	NG	NG	NG	NG	NG	OK[Notes 2, 6]
sfrp	NG	NG	NG	NG	NG	NG	NG	NG	OK[Notes 2, 7]
addr16 [Note 8]	OK	OK	OK	OK	OK	OK	OK	OK	NG
addr5	OK	OK	OK	OK	OK	OK	OK	OK	NG
bit	OK	OK	NG	NG	NG	NG	NG	NG	NG
n	OK	OK	NG	NG	NG	NG	NG	NG	NG

< Table Explanation >

Forward : This means forward reference.

Backward : This means backward reference.

OK : This means that description is possible.

NG : This means an error.

- : This means that description is not possible.

Notes 1. The defined symbol specifying sfr or sfrp (sfr area where saddr and sfr are not overlapped) as an operand of EQU directive is only referenced backward. Forward reference is prohibited.

Notes 2. If an sfr reserved word in the saddr area has been described for an instruction in which a combination of sfr / sfrp changed from saddr / saddrp exists in the operand combination, a code is output as saddr / saddrp.

Notes 3. sfr reserved word in saddr area

Notes 4. sfrp reserved word in saddr area

Notes 5. Only absolute expressions

Notes 6. Only sfr reserved words that allow 8-bit accessing

Notes 7. Only sfr reserved words that allow 16-bit accessing

Notes 8. When the address of use prohibited area (FA80H to FADFH) as a value of addr16 is described, check is not performed.

Table 2-22 Properties of Described Symbols as Operands of Directives

Symbol Attributes		NUMBER		ADDRESS, SADDR						BIT					
Relocation Attributes		Absolute Terms		Absolute Terms		Relocatable Terms		External Reference Terms		Absolute Terms		Relocatable Terms		External Reference Terms	
Reference Direction		Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward
Directive															
ORG		OK Note 1	-	-	-	-	-	-	-	-	-	-	-	-	-
EQU Note 2		OK	-	OK	-	OK Note 3	-	-	-	OK	-	OK Note 3	-	-	-
SET		OK Note 1	-	-	-	-	-	-	-	-	-	-	-	-	-
DB	Size	OK Note 1	-	-	-	-	-	-	-	-	-	-	-	-	-
	Initial value	OK	OK	OK	OK	OK	OK	OK	OK	-	-	-	-	-	-
DW	Size	OK Note 1	-	-	-	-	-	-	-	-	-	-	-	-	-
	Initial value	OK	OK	OK	OK	OK	OK	OK	OK	-	-	-	-	-	-
DS		OK Note 4	-	-	-	-	-	-	-	-	-	-	-	-	-
BR		OK	-	-	-	-	-	-	-	-	-	-	-	-	-

< Explanation >

OK : Description possible

- : Description impossible

Notes 1. Only an absolute expression can be described.

Notes 2. An error will result if an expression including one of the following patterns is described.

- ADDRESS attribute - ADDRESS attribute
- ADDRESS attribute relational operator ADDRESS attribute
- HIGH absolute ADDRESS attribute
- LOW absolute ADDRESS attribute
- DATAPOS absolute ADDRESS attribute
- MASK absolute ADDRESS attribute
- When the operation results can be affected by optimization from the above 7 patterns.

Notes 3. A term created by the HIGH / LOW / DATAPOS / MASK operator that has a relocatable term is not allowed.

Notes 4. Refer to "3.4 (3) DS (define storage)".

CHAPTER 3 DIRECTIVES

This chapter explains the directives. Directives are instructions that direct all types of instructions necessary for the RA78K0S to perform a series of processes.

3.1 Overview

Instructions are translated into object codes (machine language) as a result of assembling, but directives are not converted into object codes in principle. Directives contain the following functions mainly :

- To facilitate description of source programs
- To initialize memory and reserve memory areas
- To provide the information required for assemblers and linkers to perform their intended processing

Table 3-1 shows the types of directives.

Table 3-1 List of Directives

Type of Directive	Directives
Segment Definition Directives	CSEG, DSEG, BSEG, ORG
Symbol Definition Directives	EQU, SET
Memory Initialization and Area Reservation Directives	DB, DW, DS, DBIT
Linkage Directives	EXTRN, EXTBIT, PUBLIC
Object Module Name Declaration Directive	NAME
Automatic Branch Instruction Selection Directive	BR
Macro Directives	MACRO, LOCAL, REPT, IRP, EXITM, ENDM
Assembly Termination Directive	END

The following sections explain the details of each directive.

In the description format of each directive, "[ ]" indicates that the parameter in square brackets may be omitted from specification, and "..." indicates the repetition of description in the same format.

3.2 Segment Definition Directives

A source module must be described in units of segments.

Segment definition directives are used to define these segments. Segments are divided into the following four types :

(1) Code segments

(2) Data segments

(3) Bit segments

(4) Absolute segments

The type of segment determines the address range in memory in which each segment will be located.

Table 3-2 shows the method of defining each segment and the memory address at which each segment is located.

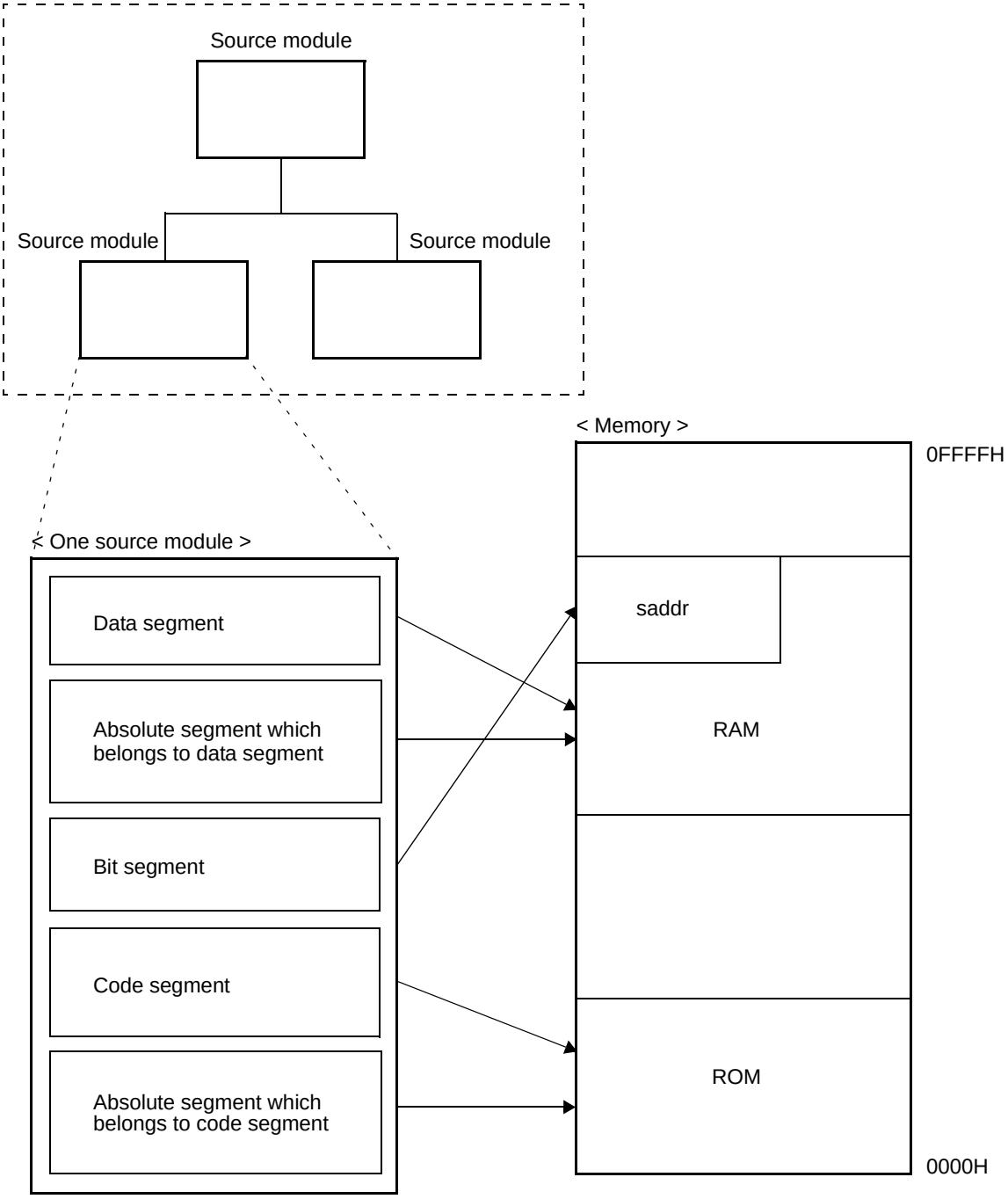
Table 3-2 Segment Definition Methods and Memory Address Location

Type of Segment	Method of Definition	Memory Address at Which Each Segment Is Located
Code segment	CSEG directive	Within the internal or external ROM address
Data segment	DSEG directive	Within the internal or external RAM address
Bit segment	BSEG directive	Within the saddr area in the internal RAM
Absolute segment	Specifies location address (AT location address) to relocation attribute with CSEG, DSEG, or BSEG directive	Specified address

If the user wishes to determine the memory location of a segment, describe the segment as an absolute segment. For the stack area, the user needs to reserve an area in the data segment and set it in the stack pointer.

An example of segment location is shown in Figure 3-1.

Figure 3-1 Memory Location of Segments



The following segment definition directives are available.

- CSEG (code segment)
- DSEG (data segment)
- BSEG (bit segment)
- ORG (origin)

CSEG

(1) CSEG (code segment)

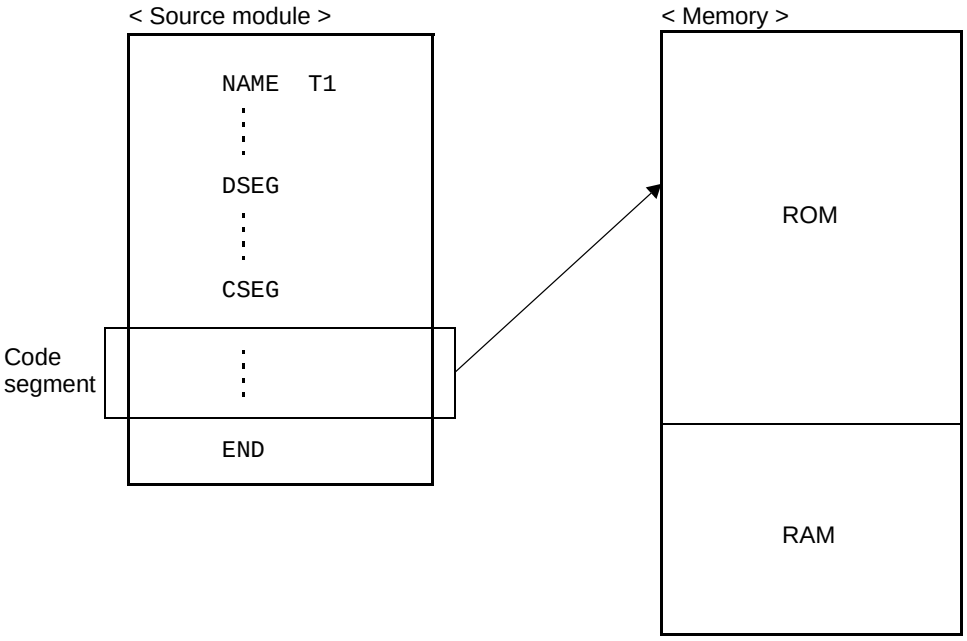
[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[segment - name]	CSEG	[relocation - attribute]	[; comment]

[Function]

- The CSEG directive indicates to the assembler the start of a code segment.
- All instructions described following the CSEG directive belong to the code segment until it comes across a Segment Definition Directives (CSEG, DSEG, BSEG, or ORG) or the END directive, and finally those instructions are located within a ROM address after being converted into machine language.

Figure 3-2 Relocation of Code Segment



[Use]

- The CSEG directive is used to describe instructions, DB, DW directives, etc. in the code segment defined by the CSEG directive. (However, to relocate the code segment from a fixed address, "AT absolute-expression" must be described as its relocation attribute in the operand field.)
- Description of one functional unit such as a subroutine should be defined as a single code segment. If the unit is relatively large or if the subroutine is highly versatile (i.e. can be utilized for development of other programs), the subroutine should be defined as a single module.

[Explanation]

- The start address of a code segment can be specified with the ORG directive. It can also be specified by describing the relocation attribute "AT absolute-expression".
- A relocation attribute defines a range of location addresses for a code segment. Relocation attributes are shown in Table 3-3.

Table 3-3 Relocation Attributes of CSEG

Relocation Attribute	Description Format	Explanation
CALLT0	CALLT0	Tells the assembler to locate the specified segment so that the start address of the segment becomes a multiple of 2 within the address range 0040H to 007FH. Specify this relocation attribute for a code segment that defines the entry address of a subroutine to be called with the one-byte instruction "CALLT".
FIXED	FIXED	Tells the assembler to locate the beginning of the specified segment within the address range 0800H to 0FFFH. Specify this relocation attribute for a code segment that defines a subroutine to be called with the 2-byte instruction "CALLF".
AT	AT absolute-expression	Tells the assembler to locate the specified segment to an absolute address (0000H to FFFFH).
UNIT	UNIT	Tells the assembler to locate the specified segment to any address (0080H to FA7FH).
UNITP	UNITP	Tells the assembler to locate the specified segment to any address, so that the start of the address may be an even number (0080H to FA7EH).
IXRAM	IXRAM	Tells the assembler to locate the specified segment to the internal expansion RAM.

- If no relocation attribute is specified for the code segment, the assembler will assume that "UNIT" has been specified.
- If a relocation attribute other than those listed in Table 3-3 is specified, the assembler will output an error message and assume that "UNIT" has been specified. An error will result if the size of each code segment exceeds that of the area specified by its relocation attribute.
- If the absolute expression specified with the relocation attribute "AT" is illegal, the assembler will output an error message and continue processing by assuming the value of the expression to be "0".

- By describing a segment name in the symbol field of the CSEG directive, the code segment can be named. If no segment name is specified for a code segment, the assembler will automatically give a default segment name to the code segment. The default segment names of the code segments are shown in Table 3-4.

Table 3-4 Default Segment Names of CSEG

Relocation Attribute	Default Segment Name
CALLT0	?CSEGT0
FIXED	?CSEGFX
UNIT (or omitted)	?CSEG
UNITP	?CSEGUP
XRAM	?CSEGIX
AT	Segment name cannot be omitted.

- An error will result if the segment name is omitted when the relocation attribute is AT.
- If two or more code segments have the same relocation attribute (except AT), these code segments may have the same segment name. These same-named code segments are processed as a single code segment within the assembler.
  An error will result if the same-named segments differ in their relocation attributes. Therefore, the number of the same-named segments for each relocation attribute is one.
- The same-named code segments in two or more different modules are combined into a single code segment at linkage.
- No segment name can be referenced as a symbol.
- The total number of segments that can be output by the assembler is up to 255 alias names, including those defined with the ORG directive. The same-named segments are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.
- The option-byte is specified within the address range 80H to 84H (differs by each device).
  An error will result if the option-byte is specified for the chip without the option-byte feature.
  When the option-byte is not specified for the chip having the option-byte feature, define a default segment of "?CSEGOB0 to ?CSEGOB4" to each address and set the initial value by reading from a device file.

[Application Examples]

```
        NAME    SAMP1
C1      CSEG            ; (1)

C2      CSEG    CALLT0  ; (2)

        CSEG    FIXED   ; (3)

C1      CSEG    CALLT0  ; (4)

        CSEG            ; (5)

        END
```

< Explanation >

(1) The assembler interprets the segment name as "C1", and the relocation attribute as "UNIT".

(2) The assembler interprets the segment name as "C2", and the relocation attribute as "CALLT0".

(3) The assembler interprets the segment name as "?CSEGFX", and the relocation attribute as "FIXED".

(4) Because the segment name "C1" was defined as the relocation attribute "UNIT" in (1), an error occurs.

(5) The assembler interprets the segment name as "?CSEG", and the relocation attribute as "UNIT".

DSEG

(2) DSEG (data segment)

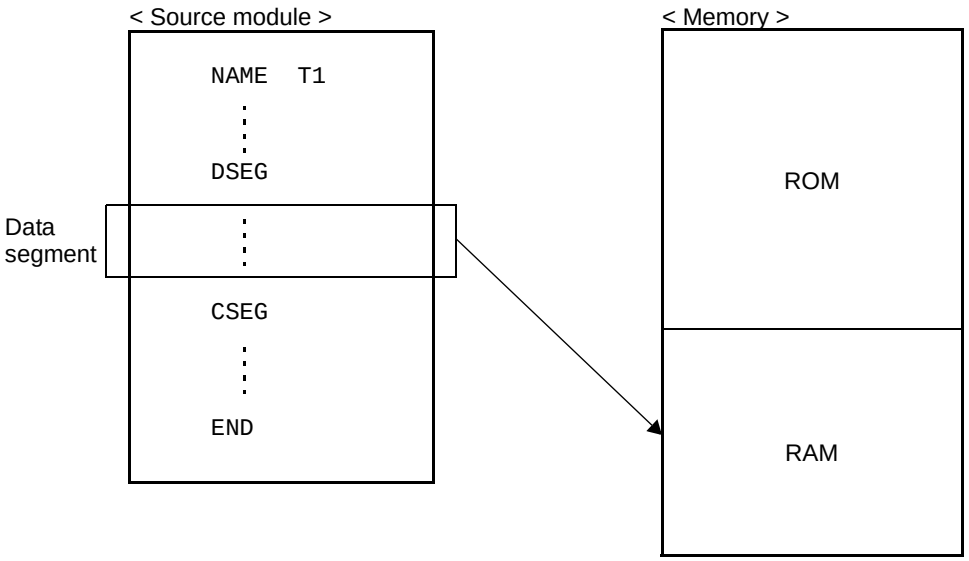
[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[segment - name]	DSEG	[relocation - attribute]	[; comment]

[Function]

- The DSEG directive indicates to the assembler the start of a data segment.
- A memory defined by the DS directive following the DSEG directive belongs to the data segment until it comes across a Segment Definition Directives (CSEG, DSEG, BSEG, or ORG) or the END directive, and finally it is reserved within the RAM address.

Figure 3-3 Relocation of Data Segment



[Use]

- The DS directive is mainly described in the data segment defined by the DSEG directive. Data segments are located within the RAM area. Therefore, no instructions can be described in any data segment.
- In a data segment, a RAM work area used in a program is reserved by the DS directive and a label is attached to each work area. Use this label when describing a source program.
  Each area reserved as a data segment is located by the linker so that it does not overlap with any other work areas on the RAM (stack area, general register area, and work areas defined by other modules).

[Explanation]

- The start address of a data segment can be specified with the ORG directive. It can also be specified by describing the relocation attribute "AT" followed by an absolute expression in the operand field of the DSEG directive.
- A relocation attribute defines a range of location addresses for a data segment. The relocation attributes available for data segments are shown in Table 3-5.

Table 3-5 Relocation Attributes of DSEG

Relocation Attribute	Description Format	Explanation
SADDR	SADDR	Tells the assembler to locate the specified segment in the saddr area (saddr area : 0FE20H to 0FEFFH).
SADDRP	SADDRP	Tells the assembler to locate the specified segment from an even-numbered address of the saddr area (saddr area : 0FE20H to 0FEFFH).
AT	AT absolute-expression	Tells the assembler to locate the specified segment in an absolute address.
UNIT	UNIT or no specification	Tells the assembler to locate the specified segment in any location (within the memory area name "RAM").
UNITP	UNITP	Tells the assembler to locate the specified segment in any location from an even-numbered address (within the memory area name "RAM").
IHRAM	IHRAM	Tells the assembler to locate the specified segment in the high-speed RAM area.
LRAM	LRAM	Tells the assembler to locate the specified segment in the low-speed RAM area.
DSPRAM	DSPRAM	Tells the assembler to locate the specified segment in the display RAM area.
IXRAM	IXRAM	Tells the assembler to locate the specified segment in the internal expansion RAM area.

- If no relocation attribute is specified for the data segment, the assembler will assume that "UNIT" has been specified.
- If a relocation attribute other than those listed in Table 3-5 is specified, the assembler will output an error message and assume that "UNIT" has been specified. An error will result if the size of each data segment exceeds that of the area specified by its relocation attribute.
- If the absolute expression specified with the relocation attribute "AT" is illegal, the assembler will output an error message and continue processing by assuming the value of the expression to be "0".

- By describing a segment name in the symbol field of the DSEG directive, the data segment can be named. If no segment name is specified for a data segment, the assembler automatically gives a default segment name. The default segment names of the data segments are shown in Table 3-6.

Table 3-6 Default Segment Names of DSEG

Relocation Attribute	Default Segment Name
SADDR	?DSEGS
SADDRP	?DSEGSP
UNIT (or no specification)	?DSEG
UNITP	?DSEGUP
IHRAM	?DSEGIH
LRAM	?DSEGL
DSPRAM	?DSEGDSP
IXRAM	?DSEGIX
AT	Segment name cannot be omitted.

- If two or more data segments have the same relocation attribute (except AT), these data segments may have the same segment name. These segments are processed as a single data segment within the assembler.
- If the relocation attribute is SADDRP, the specified segment is located so that the address immediately after the DSEG directive is described becomes a multiple of 2.
- An error occurs if the same-named segments differ in their relocation attributes. Therefore, the number of the same-named segments for each relocation attribute is one.
- The same-named data segments in two or more different modules are combined into a single data segment at linkage time.
- No segment name can be referenced as a symbol.
- The total number of segments that can be output by the assembler is up to 255 alias segments including those defined with the ORG directive. The same-named segments are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

[Application Examples]

```
        NAME    SAMP1
        DSEG                    ; (1)
WORK1 : DS      1
WORK2 : DS      2
    CSEG
        MOV     A , !WORK1      ; (2)
        MOV     A , WORK1       ; (3)
        MOVW    DE , #WORK2     ; (4)
        MOVW    AX , WORK2      ; (5)
    END
```

< Explanation >

(1) The start of a data segment is defined with the DSEG directive. Because its relocation attribute is omitted, "UNIT" is assumed. The default segment name is "?DSEG".

(2) This description corresponds to "MOV A, !addr16".

(3) This description corresponds to "MOV A, saddr". Relocatable label "WORK1" cannot be described as "saddr". Therefore, an error occurs as a result of this description.

(4) This description corresponds to "MOVW rp, #word".

(5) This description corresponds to "MOVW AX, saddrp".
Relocatable label "WORK2" cannot be described as "saddrp". Therefore, an error occurs as a result of this description.

BSEG

(3) BSEG (bit segment)

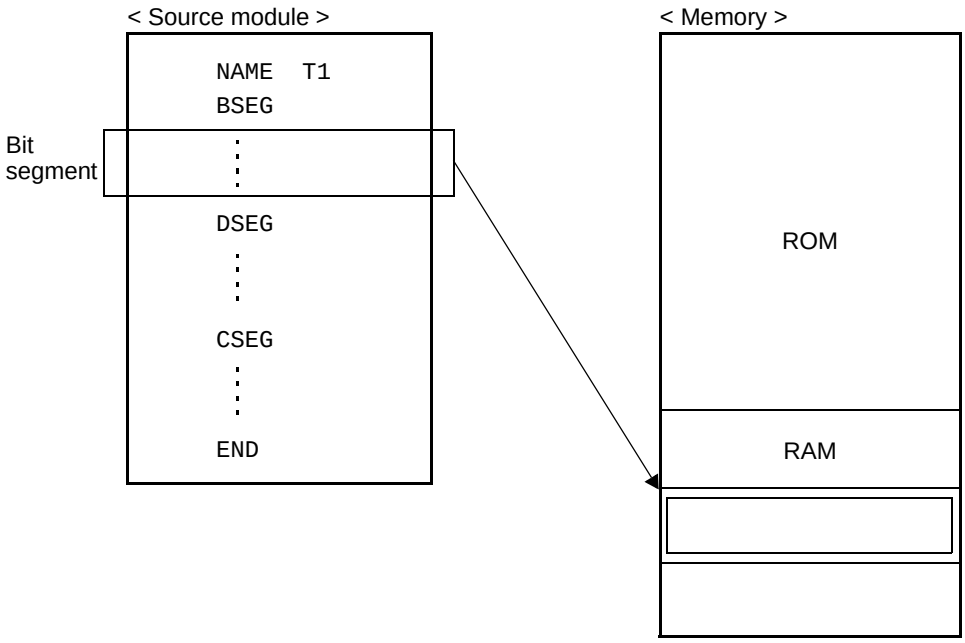
[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[segment - name]	BSEG	[relocation - attribute]	[; comment]

[Function]

- The BSEG directive indicates to the assembler the start of a bit segment.
- A bit segment is a segment that defines the RAM addresses to be used in the source module.
- A memory area that is defined by the DBIT directive after the BSEG directive until it comes across a Segment Definition Directives (CSEG, DSEG, or BSEG) or the END directive belongs to the bit segment.

Figure 3-4 Relocation of Bit Segment



[Use]

- Describe the DBIT directive in the bit segment defined by the BSEG directive (see [Application Example]).
- No instructions can be described in any bit segment.

[Explanation]

- The start address of a bit segment can be specified by describing "AT absolute-expression" in the relocation attribute field.
- A relocation attribute defines a range of location addresses for a bit segment. Relocation attributes available for bit segments are shown in Table 3-7.

Table 3-7 Relocation Attributes of BSEG

Relocation Attribute	Description Format	Explanation
AT	AT absolute-expression	Tells the assembler to locate the starting address of the specified segment in the 0th bit of an absolute address. Specification in bit units is prohibited (FE20H to FEFFH).
UNIT	UNIT (or no specification)	Tells the assembler to locate the specified segment in any location (FE20H to FEFFH).

- If no relocation attribute is specified for the bit segment, the assembler assumes that "UNIT" is specified.
- If a relocation attribute other than those listed in Table 3-7 is specified, the assembler outputs an error message and assumes that "UNIT" is specified. An error occurs if the size of each bit segment exceeds that of the area specified by its relocation attribute.
- In both the assembler and the linker, the location counter in a bit segment is displayed in the form "0xxxx.b" (The byte address is hexadecimal 4 digits and the bit position is hexadecimal 1 digit (0 to 7)).

(1) With absolute bit segment

Table 3-8 Location Counter (Absolute)

Byte address	Bit position							
	0	1	2	3	4	5	6	7
0FE20H	0FE20H.0	0FE20H.1	0FE20H.2	0FE20H.3	0FE20H.4	0FE20H.5	0FE20H.6	0FE20H.7
0FE21H	0FE21H.0	0FE21H.1	0FE21H.2	0FE21H.3	0FE21H.4	0FE21H.5	0FE21H.6	0FE21H.7

(2) With relocatable bit segment

Table 3-9 Location Counter (Relocatable)

Byte address	Bit position							
	0	1	2	3	4	5	6	7
0H	0H.0	0H.1	0H.2	0H.3	0H.4	0H.5	0H.6	0H.7
1H	1H.0	1H.1	1H.2	1H.3	1H.4	1H.5	1H.6	1H.7

Remark Within a relocatable bit segment, the byte address specifies an offset value in byte units from the beginning of the segment.
In a symbol table output by the object converter, a bit offset from the beginning of an area where a bit is defined is displayed and output.

Table 3-10 The Symbol Value and The Bit Offset display

Symbol Value	Bit Offset
00FE20H.0	0000
00FE20H.1	0001
00FE20H.2	0002
:	:
00FE20H.7	0007
00FE21H.0	0008
00FE21H.1	0009
:	:
00FE80H.0	0300
:	:

- If the absolute expression specified with the relocation attribute "AT" is illegal, the assembler outputs an error message and continues processing while assuming the value of the expression to be "0".
- By describing a segment name in the symbol field of the BSEG directive, the bit segment can be named.
  If no segment name is specified for a bit segment, the assembler automatically gives a default segment name. The following table shows the default segment names.

Table 3-11 Default Segment Names of BSEG

Relocation Attribute	Default Segment Name
UNIT (or no specification)	?BSEG
AT	Segment name cannot be omitted.

- If the relocation attribute is "UNIT", two or more data segments can have the same segment name (except AT). These segments are processed as a single segment within the assembler.
  Therefore, the number of same-named segments for each relocation attribute is one.
- The same-named bit segments in two or more different modules will be combined into a single bit segment at linkage time.
- No segment name can be referenced as a symbol.
- The only instructions that can be described in the bit segments are the DBIT, EQU, SET, PUBLIC, EXTBIT, EXTRN, MACRO, REPT, IRP, ENDM directive, macro definition and macro reference. Description of instructions other than these causes in an error.
- The total number of segments that the assembler outputs is up to 255 alias segments, with segments defined by the ORG directive. The segments having the same name are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

[Application Example]

```
        NAME    SAMP1

FLAG    EQU     0FE20H
FLAG0   EQU     FLAG.0          ; (1)
FLAG1   EQU     FLAG.1          ; (2)

        BSEG                    ; (3)
FLAG2   DBIT

        CSEG
        SET1    FLAG0           ; (4)
        SET1    FLAG2           ; (5)

        END
```

< Explanation >

(1) Bit addresses (bits 0 of 0FE20H) are defined with consideration given to byte address boundaries.

(2) Bit addresses (bits 1 of 0FE20H) are defined with consideration given to byte address boundaries.

(3) A bit segment is defined with the BSEG directive.

Because its relocation attribute is omitted, the relocation attribute "UNIT" and the segment name "?BSEG" are assumed. In each bit segment, a bit work area is defined for each bit with the DBIT directive. A bit segment should be described at the early part of the module body. Bit address FLAG2 defined within the bit segment is located without considering the byte address boundary.

(4) This description can be replaced with "SET1 FLAG.0". This FLAG indicates a byte address.

(5) In this description, no consideration is given to byte address boundaries.

ORG

(4) ORG (origin)

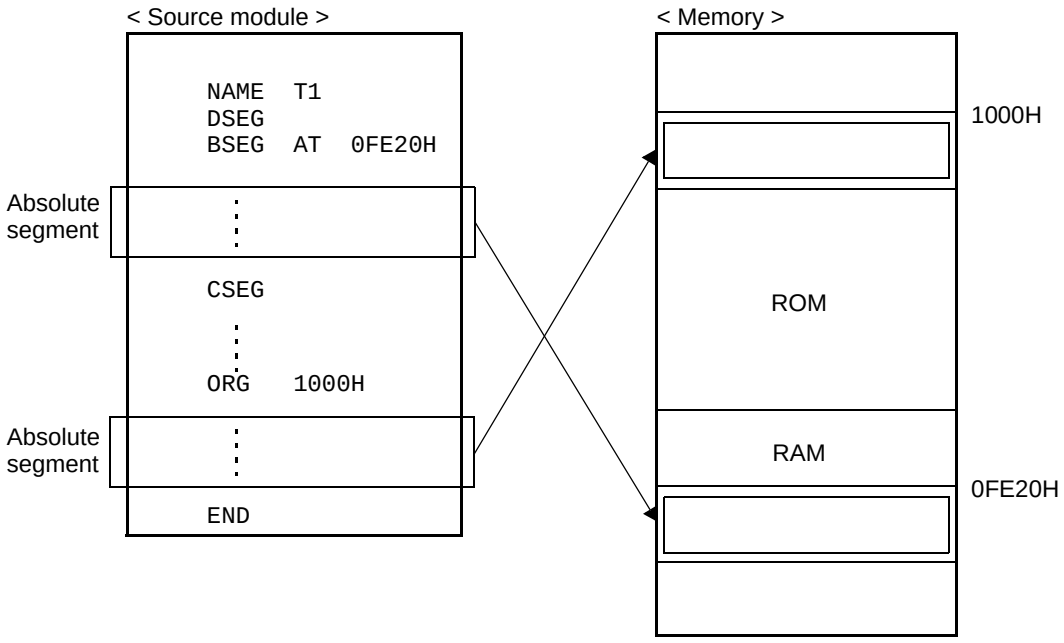
[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[segment - name]	ORG	[relocation - attribute]	[; comment]

[Function]

- The ORG directive sets the value of the expression specified by its operand of the location counter.
- After the ORG directive, described instructions or reserved memory area belongs to an absolute segment until it comes across a Segment Definition Directives (CSEG, DSEG, BSEG, or ORG) or the END directive, and they are located from the address specified by an operand.

Figure 3-5 Location of Absolute Segment



[Use]

- Specify the ORG directive to locate a code segment or data segment from a specific address.

[Explanation]

- The absolute segment defined with the ORG directive belongs to the code segment or data segment defined with the CSEG or DSEG directive immediately before this ORG directive.
  Within an absolute segment that belongs to a data segment, no instructions can be described.
  An absolute segment that belongs to a bit segment cannot be described with the ORG directive.
- The code segment or data segment defined with the ORG directive is interpreted as a code segment or data segment of the relocation attribute "AT".

- By describing a segment name in the symbol field of the ORG directive, the absolute segment can be named. The maximum number of characters that can be recognized as a segment name is 8.
- If no segment name is specified for an absolute segment, the assembler will automatically assign the default segment name "?A00xxxx", where "xxxx" indicates the four-digit hexadecimal start address (0000 to FFFF) of the segment specified.
- If neither CSEG nor DSEG directive has been described before the ORG directive, the absolute segment defined by the ORG directive is interpreted as an absolute segment in a code segment.
- If a name or label is described as the operand of the ORG directive, the name or label must be an absolute term that has already been defined in the source module.
- No segment name can be referenced as a symbol.
- The total number of segments that the assembler outputs is up to 255 alias segments, with segments defined by the Segment Definition Directives. The segments having the same name are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

[Application examples]

```
        NAME    SAMP1

        DSEG
        ORG     0FE20H          ; (1)
SADR1 : DS      1
SADR2 : DS      1
SADR3 : DS      2

MAIN0   ORG     100H
        MOV     A , SADR1       ; (2)

        CSEG                    ; (3)
MAIN1   ORG     1000H           ; (4)
        MOV     A , SADR2
        MOVW    AX , SADR3
        END
```

< Explanation >

(1) An absolute segment that belongs to a data segment is defined. This absolute segment will be located from the short direct addressing area that starts from address "FE20H".

Because specification of the segment name is omitted, the assembler automatically assigns the name "?A00FE20".

(2) Because no instruction can be described within an absolute segment that belongs to a data segment, an error occurs.

(3) This directive declares the start of a code segment.

(4) This absolute segment is located in an area that starts from address "1000H".

3.3 Symbol Definition Directives

Symbol definition directives assign names to numerical data to be used for describing a source module. These names clarify the meaning of each data value and make the contents of the source module easy to understand.

Symbol definition directives inform the assembler of the value of each name to be used in the source module.

Two directives EQU (equate) and SET (set) are available for symbol definition.

The following symbol definition directives are available :

- EQU (equate)
- SET (set)

EQU

(1) EQU (equate)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
name	EQU	expression	[; comment]

[Function]

- The EQU directive defines a name that has the value and attributes (symbol attribute and relocation attribute) of the expression specified in the operand field.

[Use]

- Define numerical data to be used in the source module as a name with the EQU directive and describe the name in the operand of an instruction in place of the numerical data.

  Numerical data to be frequently used in the source module is recommended to be defined as a name. If you must change a data value in the source module, all you need to do is to change the operand value of the name.

[Explanation]

- When a name or label is to be described in the operand of the EQU directive, use the name or label that has already been defined in the source module.

  No external reference term can be described as the operand of this directive.
- An expression including a term created by a HIGH / LOW / DATAPOS / BITPOS operator that has a relocatable term in its operand cannot be described.
- If an expression with any of the following patterns of operands is described, an error will result :
  - (1) Expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute
  - (2) Expression 1 with ADDRESS attributeRelational operatorExpression 2 with ADDRESS attribute
  - (3) Either of the following conditions (1) and (2) is fulfilled in the above expression (a) or (b) :
    - (a) If label 1 in the expression 1 with ADDRESS attribute and label 2 in the expression 2 with ADDRESS attribute belong to the same segment and if a BR directive for which the number of bytes of the object code cannot be determined is described between the two labels
    - (b) If label 1 and label 2 differ in segment and if a BR directive for which the number of bytes of the object code cannot be determined is described between the beginning of the segment and label
  - (4) HIGH absolute expression with ADDRESS attribute
  - (5) LOW absolute expression with ADDRESS attribute
  - (6) DATAPOS absolute expression with ADDRESS attribute
  - (7) BITPOS absolute expression with ADDRESS attribute

(8) The following (a) is fulfilled in the expression (4), (5), (6), or (7) :

(a) If a BR directive for which the number of bytes of the object code cannot be determined instantly is described between the label in the expression with ADDRESS attribute and the beginning of the segment to which the label belongs

- If an error exists in the description format of the operand, the assembler will output an error message, but will attempt to store the value of the operand as the value of the name described in the symbol field to the extent that it can analyze.
- A name defined with the EQU directive cannot be redefined within the same source module.
- A name that has defined a bit value with the EQU directive will have an address and bit position as value.
- Table 3-12 shows the bit values that can be described as the operand of the EQU directive and the range in which these bit values can be referenced.

Table 3-12 Representation Formats of Operands Indicating Bit Values

Operand Type	Symbol Value	Reference Range
A.bit[Note 1]	1.bit	Can be referenced within the same module only.
PSW.bit[Note 1]	1FEH.bit	
sfr[Note 2].bit[Note 1]	0FFxxH[Note 3].bit	
saddr.bit[Note 1]	0xxxxH[Note 4].bit	Can be referenced from another module.
expression.bit[Note 1]	0xxxxH[Note 4].bit	

Notes 1. 1bit = 0 to 7

Notes 2. For a detailed description, refer to the user's manual of each device.

Notes 3. "0FFxxH" indicates the address of an sfr.

Notes 4. "0xxxxH" indicates the saddr area (FE20H to FF1FH).

[Application Example]

```
        NAME    SAMP1

WORK1   EQU     0FE20H      ; (1)
WORK10  EQU     WORK1.0     ; (2)
P02     EQU     P0.2        ; (3)
A4      EQU     A.4         ; (4)
PSW5    EQU     PSW.5       ; (5)

        SET1    WORK10      ; (6)
        SET1    P02         ; (7)
        SET1    A4          ; (8)
        SET1    PSW5        ; (9)

        END
```

< Explanation >

(1) The name "WORK1" has the value "0FE20H", symbol attribute "NUMBER", and relocation attribute "ABSOLUTE".

(2) The name "WORK10" is assigned to bit value "WORK1.0", which is in the operand format "saddr.bit". "WORK1", which is described in an operand, is already defined at the value "0FE20H", in (1) above.

(3) The name "P02" is assigned to the bit value "P0.2", which is in the operand format "sfr.bit".

(4) The name "A4" is assigned to the bit value "A.4", which is in the operand format "A.bit".

(5) The name "PSW5" is assigned to the bit value "PSW.5", which is in the operand format "PSW.bit".

(6) This description corresponds to "SET1 saddr.bit".

(7) This description corresponds to "SET1 sfr.bit".

(8) This description corresponds to "SET1 A.bit".

(9) This description corresponds to "SET1 PSW.bit".

Names that have defined "sfr.bit", "A.bit", and "PSW.bit" as in (3) through (5) can be referenced only within the same module.

A name that has defined "saddr.bit" can also be referenced from another module as an external definition symbol (see "3.5 (2) EXTBIT (external bit)").

As a result of assembling the source module in example, the following assemble list is generated.

< Assemble list >

```
        Assemble list

ALNO    STNO    ADRS    OBJECT      M I SOURCE STATEMENT

   1       1                                NAME    SAMP
   2       2
   3       3            ( FE20 )            WORK1   EQU     0FE20H          ; (1)
   4       4            ( FE20.0 )          WORK10  EQU     WORK1.0         ; (2)
   5       5            ( FF00.2 )          P02     EQU     P0.2            ; (3)
   6       6            ( 0001.4 )          A4      EQU     A.4             ; (4)
   7       7            ( 01FE.5 )          PSW5    EQU     PSW.5           ; (5)
   8       8    0000    0A20                        SET1    WORK10          ; (6)
   9       9    0002    2A00                        SET1    P02             ; (7)
  10      10    0004    61CA                        SET1    A4              ; (8)
  11      11    0006    5A1E                        SET1    PSW5            ; (9)
  12      12
  13      13                                       END
```

< Explanation >

On lines (2) through (5) of the assemble list, the bit address values of the bit values defined as names are indicated in the object code field.

SET

(2) SET (set)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
name	SET	absolute - expression	[; comment]

[Function]

- The SET directive defines a name that has the value and attributes (symbol attribute and relocation attribute) of the expression specified in the operand field.
- The value and attribute of a name defined with the SET directive can be redefined within the same module. These values and attribute are valid until the same name is redefined.

[Use]

- Define numerical data (a variable) to be used in the source module as a name and describe it in the operand of an instruction in place of the numerical data (a variable).
  If you wish to change the value of a name in the source module, a different value can be defined for the same name using the SET directive again.

[Explanation]

- An absolute expression must be described in the operand field of the SET directive.
- The SET directive may be described anywhere in a source program. However, a name that has been defined with the SET directive cannot be forward-referenced.
- If an error is detected in the statement in which a name is defined with the SET directive, the assembler outputs an error message but will attempt to store the value of the operand as the value of the name described in the symbol field to the extent that it can analyze.
- A symbol defined with the EQU directive cannot be redefined with the SET directive.
  A symbol defined with the SET directive cannot be redefined with the EQU directive.
- A bit symbol cannot be defined.

[Application Example]

```
        NAME    SAMP1

COUNT   SET     10H             ; (1)

        CSEG
        MOV     B , #COUNT      ; (2)
LOOP :
        DEC     B
        BNZ     $LOOP

COUNT   SET     20H             ; (3)

        MOV     B , #COUNT      ; (4)
        END
```

< Explanation >

(1) The name "COUNT" has the value "10H", the symbol attribute "NUMBER", and relocation attribute "ABSOLUTE". The value and attributes are valid until they are redefined by the SET directive in (3) below.

(2) The value "10H" of the name "COUNT" is transferred to register B.

(3) The value of the name "COUNT" is changed to "20H".

(4) The value "20H" of the name "COUNT" is transferred to register B.

3.4 Memory Initialization and Area Reservation Directives

Memory initializing directives define the constant data to be used in a source program.

The values of the defined constant data are generated as object codes.

Area reservation directives reserve memory areas to be used in a program.

The following memory initialization and area reservation directives are available :

- DB (define byte)
- DW (define word)
- DS (define storage)
- DBIT (define bit)

DB

(1) DB (define byte)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	DB	(size)	[; comment]
		or	
[label :]	DB	initial - value [, ...]	[; comment]

[Function]

- The DB directive tells the assembler to initialize a byte area. The number of bytes to be initialized can be specified as "size".
- The DB directive also tells the assembler to initialize a memory area in byte units with the initial value(s) specified in the operand field.

[Use]

- Use the DB directive when defining an expression or character string used in the program.

[Explanation]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.
- The DB directive cannot be described in a bit segment.
  (1) With size specification :
    (a) If a size is specified in the operand field, the assembler initializes an area equivalent to the specified number of bytes with the value "00H".
    (b) An absolute expression must be described as a size. If the size description is illegal, the assembler outputs an error message and will not execute initialization.
  (2) With initial value specification :
    (a) Expression
    The value of an expression must be 8-bit data. Therefore, the value of the operand must be in the range of 0H to 0FFH. If the value exceeds 8 bits, the assembler will use only lower 8 bits of the value as valid data and output an error message.
    (b) Character string
    If a character string is described as the operand, an 8-bit ASCII code will be reserved for each character in the string.
- Two or more initial values may be specified within a statement line of the DB directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.

[Application Example]

```
                NAME    SAMP1
                CSEG
WORK1 :         DB      ( 1 )                   ; (1)
WORK2 :         DB      ( 2 )                   ; (1)
                CSEG
MASSAG :        DB      ' ABCDEF '              ; (2)
DATA1 :         DB      0AH , 0BH , 0CH         ; (3)
DATA2 :         DB      ( 3 + 1 )               ; (4)
DATA3 :         DB      ' AB ' + 1              ; (5)

                END
```

< Explanation >

(1) Because the size is specified, the assembler will initialize each byte area with the value "00H".

(2) A 6-byte area is initialized with character string 'ABCDEF'.

(3) A 3-byte area is initialized with "0AH, 0BH, 0CH".

(4) A 4-byte area is initialized with "00H".

(5) Because the value of expression "AB" + 1 is 4143H (4142H + 1) and exceeds the range of 0 to 0FFH, this description will result in an error.

DW

(2) DW (define word)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	DB	(size)	[; comment]
		or	
[label :]	DB	initial - value [, ...]	[; comment]

[Function]

- The DW directive tells the assembler to initialize a word area. The number of words to be initialized can be specified as "size".
- The DW directive also tells the assembler to initialize a memory area in word units (2 bytes) with the initial value(s) specified in the operand field.

[Use]

- Use the DW directive when defining a 16-bit numeric constant such as an address or data used in the program.

[Explanation]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified ; otherwise an initial value is assumed.
- The DW directive cannot be described in a bit segment.
  (1) With size specification :
    (a) If a size is specified in the operand field, the assembler will initialize an area equivalent to the specified number of words with the value "00H".
    (b) An absolute expression must be described as a size. If the size description is illegal, the assembler outputs an error message and will not execute initialization.
  (2) With initial value specification :
    (a) Constant
    16 bits or less.
    (b) Expression
    The value of an expression must be stored as a 16-bit data.
    No character string can be described as an initial value.
- The upper 2 digits of the specified initial value are stored in the HIGH address and the lower 2 digits of the value in the LOW address.
- Two or more initial values may be specified within a statement line of the DW directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.

[Application Example]

```
        NAME    SAMP1
        CSEG
WORK1 : DW      ( 10 )          ; (1)
WORK2 : DW      ( 128 )         ; (1)
        CSEG
        ORG     10H
        DW      MAIN            ; (2)
        DW      SUB1            ; (2)

        CSEG
MAIN :
        CSEG
SUB1 :

DATA :  DW      1234H , 5678H   ; (3)

        END
```

< Explanation >

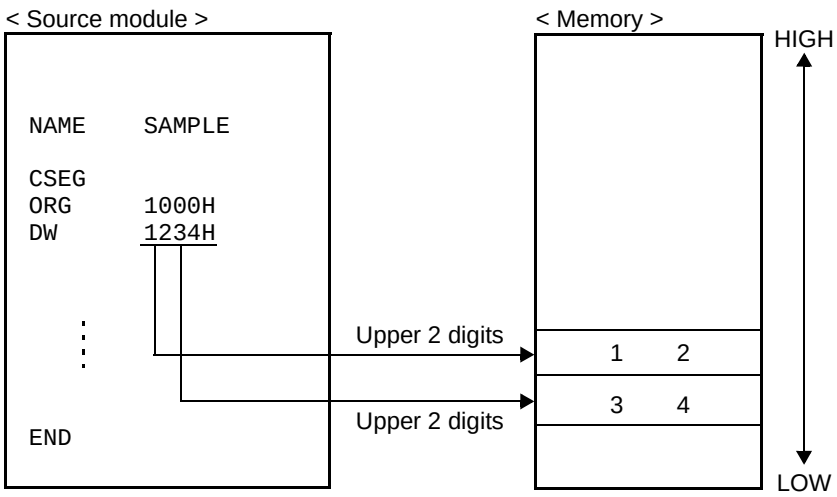
(1) Because the size is specified, the assembler will initialize each word with the value "00H".

(2) Vector entry addresses are defined with the DW directives.

(3) A 2-word area is initialized with value "34127856".

Remark The HIGH address of memory is initialized with the upper 2 digits of the word value. The LOW address of memory is initialized with the lower 2 digits of the word value.

< Example >



DS

(3) DS (define storage)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	DS	absolute - expression	[; comment]

[Function]

- The DS directive tells the assembler to reserve a memory area for the number of bytes specified in the operand field.

[Use]

- The DS directive is mainly used to reserve a memory (RAM) area to be used in the program. If a label is specified, the value of the first address of the reserved memory area is assigned to the label. In the source module, this label is used for description to manipulate the memory.

[Explanation]

- The contents of an area to be reserved with this DS directive are unknown (indefinite).
- The specified absolute expression will be evaluated with unsigned 16 bits.
- When the operand value is "0", no area can be reserved.
- The DS directive cannot be described within a bit segment.
- The symbol (label) defined with the DS directive can be referenced only in the backward direction.
- Only the following parameters extended from an absolute expression can be described in the operand field :
  - (1) A constant
  - (2) An expression with constants in which an operation is to be performed (constant expression)
  - (3) EQU symbol or SET symbol defined with a constant or constant expression
  - (4) Expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute
    If both label 1 in "expression 1 with ADDRESS attribute" and label 2 in "expression 2 with ADDRESS attribute" are relocatable, both labels must be defined in the same segment.
    However, an error will result in either of the following two cases :
  - (5) If label 1 and label 2 belong to the same segment and if a BR directive for which the number of bytes of the object code cannot be determined is described between the two labels
  - (6) If label 1 and label 2 differ in segment and if a BR directive for which the number of bytes of the object code cannot be determined is described between either label and the beginning of the segment to which the label belongs
  - (7) Any of the expressions (1) through (4) above on which an operation is to be performed.

- The following parameters cannot be described in the operand field :
  (1) External reference symbol
  (2) Symbol that has defined "expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute" with the EQU directive
  (3) Location counter ($) is described in either expression 1 or expression 2 in the form of "expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute"
  (4) Symbol that defines with the EQU directive an expression with the ADDRESS attribute on which the HIGH / LOW / DATAPOS / BITPOS operator is to be operated

[Application Example]

```
                NAME    SAMPLE
                DSEG
TABLE1 :        DS      10              ; (1)
WORK1 :         DS      1               ; (2)
WORK2 :         DS      2               ; (3)
                CSEG
                MOVW    HL , #TABLE1
                MOV     A , !WORK1
                MOVW    BC , #WORK2
                END
```

< Explanation >

(1) A 10-byte working area is reserved, but the contents of the area are unknown (indefinite). Label "TABLE1" is allocated to the start of the address.

(2) A 1-byte working area is reserved.

(3) A 2-byte working area is reserved.

DBIT

(4) DBIT (define bit)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[name]	DBIT	None	[; comment]

[Function]

- The DBIT directive tells the assembler to reserve a 1-bit memory area within a bit segment.

[Use]

- Use the DBIT directive to reserve a bit area within a bit segment.

[Explanation]

- The DBIT directive is described only in a bit segment.
- The contents of a 1-bit area reserved with the DBIT directive are unknown (indefinite).
- If a name is specified in the Symbol field, the name has an address and a bit position as its value.
- The defined name can be described at the place where saddr.bit is required.

[Application Example]

```
        NAME    SAMPLE
        BSEG
BIT1    DBIT            ; (1)
BIT2    DBIT            ; (1)
BIT3    DBIT            ; (1)

        CSEG
        SET1    BIT1    ; (2)

        CLR1    BIT2    ; (3)

        END
```

< Explanation >

(1) By these three DBIT directives, the assembler will reserve three 1-bit areas and define names (BIT1, BIT2, and BIT3) each having an address and a bit position as its value.

(2) This description corresponds to "SET1 saddr.bit" and describes the name "BIT1" of the bit area reserved in (1) above as operand "saddr.bit".

(3) This description corresponds to "CLR1 saddr.bit" and describes name "BIT2" as "saddr.bit".

3.5 Linkage Directives

Linkage directives clarify the relativity to reference a symbol defined in the other modules.

Consider a case where a program is created by being divided into two modules : Module 1 and Module 2. In Module 1, when a symbol defined in Module 2 is referenced, the symbol cannot be used without declaration in each module. For this reason, some sort of signal or indication such as "I want to use the symbol" or "You may use the symbol" is required to be issued between the two modules.

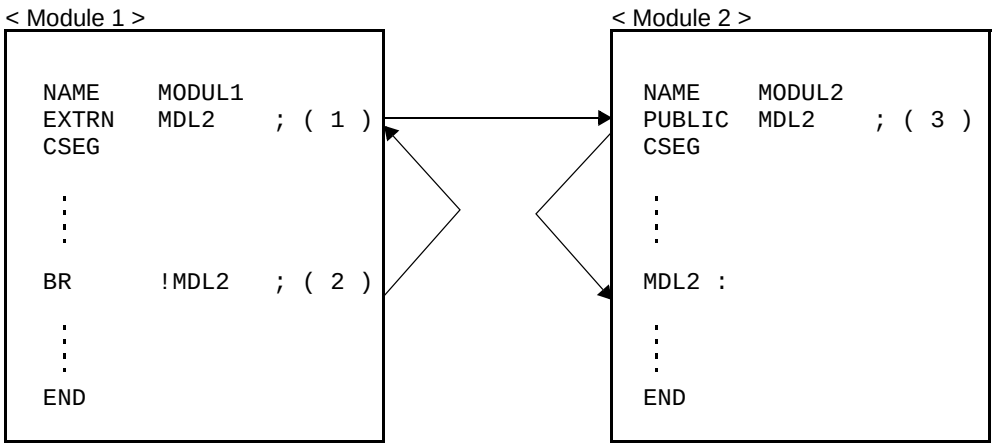
In Module 1, the external reference declaration of a symbol issues to indicate that a symbol defined in another module must be referenced. In Module 2, the external definition declaration of a symbol issues to indicate that the defined symbol may be referenced in another module.

The symbol can be referenced for the first time when both the external reference declaration and the external definition declaration are effectively made.

Linkage directives function to establish this interrelationship and are available in the following two types :

- To declare external reference of a symbol : EXTRN (external) and EXTBIT (external bit) directives
- To declare external definition of a symbol : PUBLIC (public) directive

Figure 3-6 Relationship of Symbols Between Two Modules



In module 1 in Figure 3-6, the symbol "MDL2" defined in module 2 is referenced in (2). Therefore, the symbol is declared as an external reference with the EXTRN directive in (1).

In module 2, the symbol "MDL2" to be referenced from module 1 is declared as an external definition with the PUBLIC directive in (3).

The linker checks whether or not this external reference of the symbol corresponds to the external definition of the symbol.

The following linkage directives are available :

- EXTRN (external)
- EXTBIT (external bit)
- PUBLIC (public)

EXTRN

(1) EXTRN (external)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	EXTRN	symbol - name [, ...]	[; comment]

[Function]

- The EXTRN directive declares to the linker that a symbol (other than bit symbols) in another module is to be referenced in this module.

[Use]

- When referencing a symbol defined in another module, the EXTRN directive must be used to declare the symbol as an external reference.

[Explanation]

- The EXTRN directive may be described anywhere in a source program (see "2.1 Basic Configuration").
- Up to 20 symbols can be specified in the operand field by delimiting each symbol name with a comma (,).
- When referencing a symbol having a bit value, the symbol must be declared as an external reference with the EXTBIT directive.
- The symbol declared with the EXTRN directive must be declared in another module with a PUBLIC directive.
- No macro name can be described as the operand of EXTRN directive (see "CHAPTER 5 MACROS" for the macro name).
- The EXTRN directive enables only one EXTRN declaration for a symbol in an entire module. For the second and subsequent EXTRN declarations for the symbol, the linker will output a warning message.
- A symbol that has been declared cannot be described as the operand of the EXTRN directive. Conversely, a symbol that has been declared as EXTRN cannot be redefined or declared with any other directive.
- A symbol defined by the EXTRN directive can be used to for reference of an saddr area.

[Application Example]

< Module 1 >

```
        NAME    SAMP1
        EXTRN   SYM1 , SYM2     ; (1)
        CSEG
S1 :    DW      SYM1            ; (2)
        MOV     A , SYM2        ; (3)
        END
```

< Module 2 >

```
        NAME    SAMP2
        PUBLIC  SYM1 , SYM2     ; (4)
        CSEG
SYM1    EQU     0FFH            ; (5)
DATA1   DSEG    SADDR
SYM2 :  DB      012H            ; (6)
        END
```

< Explanation >

(1) This EXTRN directive declares symbols "SYM1" and "SYM2" to be referenced in (2) and (3) as external references. Two or more symbols may be described in the operand field.

(2) This DW instruction references symbol "SYM1".

(3) This MOV instruction references symbol "SYM2" and outputs a code that references an saddr area.

(4) The symbols "SYM1" and "SYM2" are declared as external definitions.

(5) The symbol "SYM1" is defined.

(6) The symbol "SYM2" is defined.

EXTBIT

(2) EXTBIT (external bit)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	EXTBIT	bit - symbol - name [, ...]	[; comment]

[Function]

- The EXTBIT directive declares to the linker that a bit symbol that has a value of saddr.bit in another module is to be referenced in this module.

[Use]

- When referencing a symbol that has a bit value and has been defined in another module, the EXTBIT directive must be used to declare the symbol as an external reference.

[Explanation]

- The EXTBIT directive may be described anywhere in a source program.
- Up to 20 symbols can be specified in the operand field by delimiting each symbol with a comma (,).
- A symbol declared with the EXTBIT directive must be declared with a PUBLIC directive in another module.
- The EXTBIT directive enables only one EXTBIT declaration for a symbol in an entire module. For the second and subsequent EXTBIT declarations for the symbol, the linker will output a warning message.

[Application Example]

< Module 1 >

```
NAME    SAMP1
EXTBIT  FLAG1 , FLAG2           ; (1)
CSEG
SET1    FLAG1                   ; (2)
CLR1    FLAG2                   ; (3)
END
```

< Module 2 >

```
        NAME    SAMP2
        PUBLIC  FLAG1 , FLAG2   ; (4)
        BSEG
FLAG1   DBIT                    ; (5)
FLAG2   DBIT                    ; (6)
        CSEG
        NOP
        END
```

< Explanation >

(1) This EXTBIT directive declares symbols "FLAG1" and "FLAG2" to be referenced as external references. Two or more symbols may be described in the operand field.

(2) This SET1 instruction references symbol "FLAG1". This description corresponds to "SET1 saddr.bit".

(3) This CLR1 instruction references symbol "FLAG2". This description corresponds to "CLR1 saddr.bit".

(4) This PUBLIC directive defines symbols "FLAG1" and "FLAG2".

(5) This DBIT directive defines symbol "FLAG1" as a bit symbol of SADDR area.

(6) This DBIT directive defines symbol "FLAG2" as a bit symbol of SADDR area.

PUBLIC

(3) PUBLIC (public)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	PUBLIC	symbol - name [, ...]	[; comment]

[Function]

- The PUBLIC directive declares to the linker that the symbol described in the operand field is a symbol to be referenced from another module.

[Use]

- When defining a symbol (including bit symbol) to be referenced from another module, the PUBLIC directive must be used to declare the symbol as an external definition.

[Explanation]

- The PUBLIC directive may be described anywhere in a source program.
- Up to 20 symbols can be specified in the operand field by delimiting each symbol name with a comma (,).
- Symbol(s) to be described in the operand field must be defined within the same module.
- The PUBLIC directive enables only one PUBLIC declaration for a symbol in an entire module. The second and subsequent PUBLIC declarations for the symbol will be ignored by the linker.
- The following symbols cannot be used as the operand of the PUBLIC directive :
  (1) Name defined with the SET directive
  (2) Symbol defined with the EXTRN or EXTBIT directive within the same module
  (3) Segment name
  (4) Module name
  (5) Macro name
  (6) Symbol not defined within the module
  (7) Symbol defining an operand with a bit attribute with the EQU directive
  (8) Symbol defining an sfr with the EQU directive (however, the place where sfr area and saddr area are overlapped is excluded)

[Application Example]

< Module 1 >

```
        NAME    SAMP1
        PUBLIC  A1 , A2         ; (1)
        EXTRN   B1
        EXTBIT  C1

A1      EQU     10H
A2      EQU     0FE20H.1

        CSEG
        BR      B1
        SET1    C1
        END
```

< Module 2 >

```
        NAME    SAMP2
        PUBLIC  B1              ; (2)
        EXTRN   A1
        CSEG
B1 :
        MOV     C , #LOW ( A1 )
        END
```

< Module 3 >

```
        NAME    SAMP3
        PUBLIC  C1              ; (3)
        EXTBIT  A2
C1      EQU     0FE21H.0
        CSEG
        CLR1    A2
        END
```

< Explanation >

(1) This PUBLIC directive declares that symbols "A1" and "A2" are to be referenced from other modules.

(2) This PUBLIC directive declares that symbol "B1" is to be referenced from another module.

(3) This PUBLIC directive declares that symbol "C1" is to be referenced from another module.

3.6 Object Module Name Declaration Directive

The object module name declaration directive gives a module name to an object module to be created by the RA78K0S assembler.

The following object module name declaration directive is available :

- NAME (name)

NAME

(1) NAME (name)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	NAME	object - module - name	[; comment]

[Function]

- The NAME directive assigns the object module name described in the operand field to an object module to be output by the assembler.

[Use]

- A module name is required for each object module in symbolic debugging with a debugger.

[Explanation]

- The NAME directive may be described anywhere in a source program.
- For the conventions of module name description, see the conventions on symbol description in "2.2.3 Symbol field".
- Characters that can be specified as a module name are those characters permitted by the operating system of the assembler software other than "(" (28H) or ")" (29H) or kanji characters.
- No module name can be described as the operand of any directive other than NAME or of any instruction.
- If the NAME directive is omitted, the assembler will assume the primary name (first 8 characters) of the input source module file as the module name. In the Windows version, the primary name is converted to capital letters for retrieval. If two or more module names are specified, the assembler will output a warning message and ignore the second and subsequent module name declarations.
- A module name to be described in the operand field must not exceed eight characters.
- The uppercase and lowercase characters of a symbol name are distinguished.

[Application Example]

```
        NAME    SAMPLE  ; (1)
        DSEG
BIT1 :  DBIT

        CSEG
        MOV     A , B

        END
```

< Explanation >

(1) This NAME directive declares "SAMPLE" as a module name.

3.7 Automatic Branch Instruction Selection Directive

Unconditional branch instructions directly describe a branch destination address as their operand. Two such instructions, "BR !addr16" and "BR $addr16", are available. These instructions select and use the most appropriate operand according to the address range of the branch destination. Since the number of bytes is different for each directive, in order to create a program with high memory utilization efficiency, it is necessary to use the instruction with the smallest number of bytes. However, it is quite troublesome to take this address range into account when describing the branch instruction.

For this reason, there was a need for a directive that directs the assembler to automatically select the two-byte or three-byte branch instruction according to the address range of the branch destination. This is called automatic branch instruction selection directive.

The following automatic branch instruction selection directive is available :

- BR (branch)

BR

(1) BR (branch)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	BR	expression	[; comment]

[Function]

- The BR directive tells the assembler to automatically select a 2- or 3-byte BR branch instruction according to the value range of the expression specified in the operand field and to generate the object code applicable to the selected instruction.

[Use]

- If the branch destination is within the range of -80H to +7FH from the address next to the BR directive, the 2-byte branch instruction "BR $addr16" can be described. With this instruction, the required memory space can be reduced by one byte as compared with that when using the 3-byte branch instruction "BR !addr16". To create a program with high memory utilization efficiency, the 2-byte branch instruction should be used positively.

  However, it is troublesome to take the address range of the branch destination into account when describing the branch instruction. Therefore, use the BR directive if it is unclear whether a 2-byte branch instruction can be described.
- If it is definite that you can describe a 2-byte or 3-byte branch instruction, describe the applicable instruction. This shortens the assembly time in comparison with describing the BR directive.

[Explanation]

- The BR directive can only be used within a code segment.
- The direct jump destination is described as the operand of the BR directive. "$" indicating the current location counter at the beginning of an expression cannot be described.
- For optimization, the following conditions must be satisfied.
  (1) No more than 1 label or forward-reference symbol in the expression.
  (2) Do not describe an EQU symbol with the ADDRESS attribute.
  (3) Do not describe an EQU defined symbol for "expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute".
  (4) Do not describe an expression with ADDRESS attribute on which the HIGH / LOW / DATAPOS / BITPOS operator has been operated.

  If these conditions are not met, the 3-byte BR instruction will be selected.

[Application Example]

```
ADDRESS                 NAME    SAMPLE
                C1      CSEG    AT      50H
000050H                 BR      L1                  ; (1)
000052H                 BR      L2                  ; (2)
    :
00007DH         L1 :
    :
007FFFH         L2 :
                        END
```

< Explanation >

(1) This BR directive generates a 2-byte branch instruction (BR $addr16) because the displacement between this line and the branch destination is within the range of -80H and +7FH.

(2) This BR directive will be substituted with the 3-byte branch instruction (BR !addr16) because the displacement between this line and the branch destination is without the range of -80H and +7FH.

3.8 Macro Directives

When you describe a source program, it is troublesome to describe a series of frequently used instruction groups over and over again, and this may cause an increase in the number of description or coding errors.

By using the macro function with macro directives, the need to repeatedly describe the same group of instructions can be eliminated, thereby increasing coding efficiency of the program. The basic function of a macro is the substitution of a series of statements with a name.

The following macro directives are available :

- MACRO (macro)
- LOCAL (local)
- REPT (repeat)
- IRP (indefinite repeat)
- EXITM (exit from macro)
- ENDM (end macro)

MACRO

(1) MACRO (macro)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
macro - name	MACRO : Macro body : ENDM	[formal - parameter [, ...]]	[; comment] [; comment]

[Function]

- The MACRO directive executes a macro definition by assigning the macro name specified in the symbol field to a series of statements (called a macro body) described between this directive and the ENDM directive.

[Use]

- Define a series of frequently used statements in the source program with a macro name. After its definition only describe the defined macro name (see "5.2.2 Macro reference"), and the macro body corresponding to the macro name is expanded.

[Explanation]

- The MACRO directive must be paired with the ENDM directive.
- For the macro name to be described in the symbol field, see the conventions of symbol description in "2.2.3 Symbol field".
- To reference a macro, describe the defined macro name in the mnemonic field.
- For the formal parameter(s) to be described in the operand field, the same rules as the conventions of symbol description will apply.
- Up to 16 formal parameters can be described per macro directive.
- Formal parameters are valid only within the macro body.
- An error will result if any reserved word is described as a formal parameter. However, if a user-defined symbol is described, its recognition as a formal parameter will take precedence.
- The number of formal parameters must be the same as the number of actual parameters.
- A name or label defined within the macro body if declared with the LOCAL directive becomes effective with respect to one-time macro expansion.
- Nesting of macros (i.e., to refer to other macros within the macro body) is allowed up to eight levels including REPT and IRP directives.
- The number of macros that can be defined within a single source module is not specifically limited. In other words, macros may be defined as long as there is memory space available.
- Formal parameter definition lines, reference lines, and symbol names are not output to a cross-reference list.
- Two or more segments must not be defined in a macro body. If defined, an error will be output.

[Application Example]

```
        NAME    SAMPLE

ADMAC   MACRO   PARA1 , PARA2   ; (1)
        MOV     A , #PARA1
        ADD     A , #PARA2
        ENDM                    ; (2)

        ADMAC   10H , 20H       ; (3)

        END
```

< Explanation >

(1) A macro is defined by specifying macro name "ADMAC" and two formal parameters "PARA1" and "PARA2".

(2) This directive indicates the end of the macro definition.

(3) Macro "ADMAC" is referenced.

LOCAL

(2) LOCAL (local)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
None	LOCAL	symbol - name [, ...]	[; comment]

[Function]

- The LOCAL directive declares that the symbol name specified in the operand field is a local symbol that is valid only within the macro body.

[Use]

- If a macro that defines a symbol within the macro body is referenced more than once, the assembler will output a double definition error for the symbol. By using the LOCAL directive, you can reference (or call) a macro, which defines symbol(s) within the macro body, more than once.

[Explanation]

- For the conventions on symbol names to be described in the operand field, see the conventions on symbol description in "2.2.3 Symbol field".
- A symbol declared as LOCAL will be substituted with a symbol "??RAn" (where n = 0000 to FFFF) at each macro expansion. The symbol "??RAn" after the macro replacement will be handled in the same way as a global symbol and will be stored in the symbol table, and can thus be referenced under the symbol name "??RAn".
- If a symbol is described within a macro body and the macro is referenced more than once, it means that the symbol would be defined more than once in the source module. For this reason, it is necessary to declare that the symbol is a local symbol that is valid only within the macro body.
- The LOCAL directive can be used only within a macro definition.
- The LOCAL directive must be described before using the symbol specified in the operand field (in other words, the LOCAL directive must be described at the beginning of the macro body).
- Symbol names to be defined with the LOCAL directive within a source module must be all different (in other words, the same name cannot be used for local symbols to be used in each macro).
- The number of local symbols that can be specified in the operand field is not limited as long as they are all within a line. However, the number of symbols within a macro body is limited to 64. If 65 or more local symbols are declared, the assembler will output an error message and store the macro definition as an empty macro body. Nothing will be expanded even if the macro is called.
- Macros defined with the LOCAL directive cannot be nested.
- Symbols defined with the LOCAL directive cannot be called (referenced) from outside the macro.
- No reserved word can be described as a symbol name in the operand field. However, if a symbol same as the user-defined symbol is described, its recognition as a local symbol will take precedence.

- A symbol declared as the operand of the LOCAL directive will not be output to a cross-reference list and symbol table list.
- The statement line of the LOCAL directive will not be output at the time of the macro expansion.
- If a LOCAL declaration is made within a macro definition for which a symbol has the same name as a formal parameter of that macro definition, an error will be output.

[Application Example]

```
        NAME    SAMPLE

        ; Macro definition
MAC1    MACRO
        LOCAL   LLAB            ; (1)
LLAB :                          ;
        BR      $LLAB           ; (2)
        ENDM                    ;

        ; Source text
REF1 :  MAC1                    ; (3)

        BR      !LLAB           ; (4)   <-- This description is erroneous.

        REF2 :  MAC1            ; (5)

        END
```

< Explanation >

(1) This LOCAL directive defines symbol name "LLAB" as a local symbol.

(2) This BR instruction references local symbol "LLAB" within macro MAC1.

(3) This macro reference calls macro MAC1.

(4) Because local symbol "LLAB" is referenced outside the definition of macro MAC1, this description results in an error.

(5) This macro reference calls macro MAC1.

The assemble list of the above application example is shown below.

< Assemble list >

```
        Assemble list

ALNO    STNO    ADRS    OBJECT  M I     SOURCE STATEMENT

  1       1                                       NAME    SAMPLE
  2       2                     M       MAC1      MACRO
  3       3                     M               LOCAL   LLAB            ; (1)
  4       4                     M       LLAB :
  5       5                     M               BR      $LLAB           ; (2)
  6       6                     M               ENDM
  7       7
  8       8     000000                  REF1 :  MAC1                    ; (3)
          9                     #1      ;
         10     000000         #1      ??RA0000:
         11     000000  14FE   #1              BR      $??RA0000       ; (2)
  9      12
 10      13     000002  2C0000                 BR      !LLAB           ; (4)
***   ERROR E2407 , STNO  13 ( 0 ) Undefined symbol reference ' LLAB '
***   ERROR E2303 , STNO  13 ( 13 ) Illegal expression
 11      14
 12      15     000005                  REF2 :  MAC1                    ; (5)
         16                     #1      ;
         17     000005         #1      ??RA0001 :
         18     000005  14FE   #1              BR      $??RA0001       ; (2)
 13      19
 14      20                                      END
```

REPT

(3) REPT (repeat)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	REPT	absolute - expression	[; comment]
	:		
	ENDM		[; comment]

[Function]

- The REPT directive tells the assembler to repeatedly expand a series of statements described between this directive and the ENDM directive (called the REPT-ENDM block) the number of times equivalent to the value of the expression specified in the operand field.

[Use]

- Use the REPT and ENDM directives to describe a series of statements repeatedly in a source program.

[Explanation]

- An error occurs if the REPT directive is not paired with the ENDM directive.
- In the REPT-ENDM block, macro references, REPT directives, and IRP directives can be nested up to eight levels.
- If the EXITM directive appears in the REPT-ENDM block, subsequent expansion of the REPT-ENDM block by the assembler is terminated.
- Assembly control instructions may be described in the REPT-ENDM block.
- Macro definitions cannot be described in the REPT-ENDM block.
- The absolute expression described in the operand field is evaluated with unsigned 16 bits. If the value of the expression is 0, nothing is expanded.

[Application Example]

```
NAME    SAMP1
CSEG
        ; REPT-ENDM block
REPT    3                       ; (1)
        INC     B
        DEC     C
        ; Source text
ENDM                            ; (2)
END
```

< Explanation >

(1) This REPT directive tells the assembler to expand the REPT-ENDM block three consecutive times.

(2) This directive indicates the end of the REPT-ENDM block.

When the above source program is assembled, the REPT-ENDM block is expanded as shown in the following assemble list :

< Assemble List >

```
NAME    SAMP1
CSEG
REPT    3
        INC     B
        DEC     C
ENDM
INC     B
DEC     C
INC     B
DEC     C
INC     B
DEC     C
END
```

The REPT-ENDM block defined by statements (1) and (2) has been expanded three times. On the assemble list, the definition statements (1) and (2) by the REPT directive in the source module is not displayed.

IRP

(4) IRP (indefinite repeat)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	IRP	formal - parameter , <[actual - parameter [, ...]]>	[; comment]
	:		
	ENDM		[; comment]

[Function]

- The IRP directive tells the assembler to repeatedly expand a series of statements described between this directive and the ENDM directive (called the IRP-ENDM block) the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters specified in the operand field.

[Use]

- Use the IRP and ENDM directives to describe a series of statements, only some of which become variables, repeatedly in a source program.

[Explanation]

- The IRP directive must be paired with the ENDM directive.
- Up to 16 actual parameters may be described in the operand field.
- In the IRP-ENDM block, macro references, REPT and IRP directives can be nested up to eight levels.
- If the EXITM directive appears in the IRP-ENDM block, subsequent expansion of the IRP-ENDM block by the assembler is terminated.
- Macro definitions cannot be described in the IRP-ENDM block.
- Assembly control instructions may be described in the IRP-ENDM block.

[Application Example]

```
NAME    SAMP1
CSEG

IRP     PARA , < 0AH , 0BH , 0CH >      ; (1)
        ; IRP-ENDM block
        ADD     A , #PARA
        MOV     [ DE ] , A
ENDM                                    ; (2)
        ; Source text
END
```

< Explanation >

(1) The formal parameter is "PARA" and the actual parameters are the following three : "0AH", "0BH", and "0CH". This IRP directive tells the assembler to expand the IRP-ENDM block three times (i.e., the number of actual parameters) while replacing the formal parameter "PARA" with the actual parameters "0AH", "0BH", and "0CH".

(2) This directive indicates the end of the IRP-ENDM block.

When the above source program is assembled, the IRP-ENDM block is expanded as shown in the following assemble list :

< Assemble List >

```
NAME    SAMP1
CSEG
        ; IRP-ENDM block
ADD     A , #0AH                ; (3)
MOV     [ DE ] , A
ADD     A , #0BH                ; (4)
MOV     [ DE ] , A
ADD     A , #0CH                ; (5)
MOV     [ DE ] , A
        ; Source text
END
```

The IRP-ENDM block defined by statements (1) and (2) has been expanded three times (equivalent to the number of actual parameters).

(3) In this ADD instruction, PARA is replaced with 0AH.

(4) In this ADD instruction, PARA is replaced with 0BH.

(5) In this ADD instruction, PARA is replaced with 0CH.

EXITM

(5) EXITM (exit from macro)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	IEXITM	None	[; comment]

[Function]

- The EXITM directive forcibly terminates the expansion of the macro body defined by the MACRO directive and the repetition by the REPT-ENDM or IRP-ENDM block.

[Use]

- This function is mainly used when a conditional assembly function (see "4.7 Conditional Assembly Control Instructions") is used in the macro body defined with the MACRO directive.
- If conditional assembly functions are used in combination with other instructions in the macro body, part of the source program that must not be assembled is likely to be assembled unless control is returned from the macro by force using this EXITM directive. In such cases, be sure to use the EXITM directive.

[Explanation]

- If the EXITM directive is described in a macro body, instructions up to the ENDM directive will be stored as the macro body.
- The EXITM directive indicates the end of a macro only during the macro expansion.
- If something is described in the operand field of the EXITM directive, the assembler will output an error message but will execute the EXITM processing.
- If the EXITM directive appears in a macro body, the assembler will return by force the nesting level of IF / _IF / ELSE / ELSEIF / _ELSEIF / ENDIF blocks to the level when the assembler entered the macro body.
- If the EXITM directive appears in an INCLUDE file resulting from expanding the INCLUDE control instruction described in a macro body, the assembler will accept the EXITM directive as valid and terminate the macro expansion at that level.

[Application Example]

- In the example here, conditional assembly control instructions are used. See "4.7 Conditional Assembly Control Instructions".
- See "CHAPTER 5 MACROS" for the macro body and macro expansion.

```
        NAME    SAMP1
MAC1    MACRO                           ; (1)
        ; macro body
        NOT1    CY
$       IF ( SW1 )                      ; (2)   <-- IF block
                BT      A.1 , $L1
                EXITM                   ; (3)
$       ELSE                            ; (4)   <-- ELSE block
                MOV1    CY , A.1
                MOV     A , #0
$       ENDIF                           ; (5)
$       IF ( SW2 )                      ; (6)   <-- IF block
                BR      [ HL ]
$       ELSE                            ; (7)   <-- ELSE block
                BR      [ DE ]
$       ENDIF                           ; (8)
        ; Source text
        ENDM                            ; (9)

        CSEG
$       SET ( SW1 )                     ; (10)
        MAC1                            ; (11)  <-- Macro reference
L1 :    NOP
        END
```

< Explanation >

(1) The macro "MAC1" uses conditional assembly functions (2) and (4) through (8) within the macro body.

(2) An IF block for conditional assembly is defined here. If switch name "SW1" is true (not "0"), the ELSE block is assembled.

(3) This directive terminates by force the expansion of the macro body in (4) and thereafter.
If this EXITM directive is omitted, the assembler proceeds to the assembly process in (6) and thereafter when the macro is expanded.

(4) An ELSE block for conditional assembly is defined here. If switch name "SW1" is false ("0"), the ELSE block is assembled.

(5) This ENDIF control instruction indicates the end of the conditional assembly.

(6) Another IF block for conditional assembly is defined here. If switch name "SW2" is true (not "0"), the following IF block is assembled.

(7) Another ELSE block for conditional assembly is defined. If switch name "SW2" is false ("0"), the ELSE block is assembled.

(8) This ENDIF instruction indicates the end of the conditional assembly processes in (6) and (7).

(9) This directive indicates the end of the macro body.

(10) This SET control instruction gives true value (not "0") to switch name "SW1" and sets the condition of the conditional assembly.

(11) This macro reference calls macro "MAC1".

When the source program in the above example is assembled, macro expansion occurs as shown below.

< Assemble List >

```
        NAME    SAMP1
MAC1    MACRO                   ; (1)
            :
        ENDM                    ; (9)
        CSEG
$       SET ( SW1 )             ; (10)
        MAC1                    ; (11)
        ; Macro-expanded part
        NOT1    CY
$       IF ( SW1 )
                BT      A.1 , $L1
        ; Source text
L1 :    NOP
        END
```

The macro body of macro "MAC1" is expanded by referring to the macro in (11). Because true value is set in switch name "SW1" in (10), the first IF block in the macro body is assembled. Because the EXITM directive is described at the end of the IF block, the subsequent macro expansion is not executed.

ENDM

(6) ENDM (end macro)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
None	ENDM	None	[; comment]

[Function]

- The ENDM directive instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.

[Use]

- The ENDM directive must always be described at the end of a series of statements following the MACRO, REPT, and / or the IRP directives.

[Explanation]

- A series of statements described between the MACRO directive and ENDM directive becomes a macro body.
- A series of statements described between the REPT directive and ENDM directive becomes a REPT-ENDM block.
- A series of statements described between the IRP directive and ENDM directive becomes an IRP-ENDM block.

[Application Examples]

Example 1 < MACRO-ENDM >

```
        NAME    SAMP1
ADMAC   MACRO   PARA1 , PARA2
                MOV     A , #PARA1
                ADD     A , #PARA2
        ENDM
            :
        END
```

Example 2 < REPT-ENDM >

```
NAME    SAMP2
CSEG
    :
REPT    3
        INC     B
        DEC     C
ENDM
    :
END
```

Example 3 < IRP-ENDM >

```
NAME    SAMP3
CSEG
    :
IRP     PARA , < 1 , 2 , 3 >
        ADD     A , #PARA
        MOV     [ DE ] , A
ENDM
    :
END
```

3.9 Assembly Termination Directive

The assembly termination directive informs the assembler of the end of a source module. This assembly termination directive must always be described at the end of each source module.

The assembler processes a series of statements up to the assembly termination directive as a source module. Therefore, if the assembly termination directive exists before the ENDM in a REPT block or an IRP block, the REPT block or IRP block becomes invalid.

The following assembly termination directive is available :

- END (end)

END

(1) END (end)

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
None	END	None	[; comment]

[Function]

- The END directive indicates to the assembler the end of a source module.

[Use]

- The END directive must always be described at the end of each source module.

[Explanation]

- The assembler continues to assemble a source module until the END directive appears in the source module. Therefore, the END directive is required at the end of each source module.
- Always input a line-feed (LF) code after the END directive.
- If any statement other than blank, tab, LF, or comments appears after the END directive, the assembler outputs a warning message.

[Application Example]

```
NAME    SAMPLE
DSEG
    :
CSEG
    :
END             ; (1)
```

< Explanation >

(1) Always describe the END directive at the end of each source module.

CHAPTER 4 CONTROL INSTRUCTIONS

This chapter explains the control instructions. Control instructions provide detailed directions on the operation of the assembler.

4.1 Overview

Control instructions are described in a source program to provide detailed directions on the operation of the assembler.

These instructions are not subject to object code generation.

Control instructions are available in the following types :

Table 4-1 List of Control Instructions

Type of Control Instruction	Control Instruction
Processor Type Specification Control Instruction	PROCESSOR
Debug Information Output Control Instructions	DEBUG / NODEBUG, DEBUGA / NODEBUGA
Cross-Reference List Output Specification Control Instructions	XREF / NOXREF, SYMLIST / NOSYMLIST
Inclusion Control Instruction	INCLUDE
Assembly List Control Instructions	EJECT, LIST / NOLIST, GEN / NOGEN, COND / NOCOND, TITLE, SUBTITLE, FORMFEED / NOFORMFEED, WIDTH, LENGTH, TAB
Conditional Assembly Control Instructions	IF / _IF / ELSEIF / _ELSEIF / ELSE / ENDIF, SET / RESET
Other Control Instructions	TOL_INF, DGL, DGS

Control instructions are described in a source program in the same way as the assembler directives.

Of the control instructions listed in Table 4-1, the following instructions have the same functions as assembler options that can be specified in the start-up command line of the assembler.

The correspondence between the control instructions and the command line assembler options is given in Table 4-2.

Table 4-2 Control Instructions and Assembler Options

Control Instructions	Assembler Options
PROCESSOR	-c
DEBUG / NODEBUG	-g / -ng
DEBUGA / NODEBUGA	-ga / -nga
XREF / NOXREF	-kx / -nkx
SYMLIST / NOSYMLIST	-ks / -nks
TITLE	-lh
FORMFEED / NOFORMFEED	-lf / -nlf
WIDTH	-lw
LENGTH	-ll
TAB	-lt

For the method of specifying the control instructions and assembler options by command line, see the RA78K0S Series Assembler Package Operation User's Manual.

4.2 Processor Type Specification Control Instruction

The processor type specification control instruction specifies in a source module file the type of target device subject to assembly.

The following processor type specification control instruction is available :

- PROCESSOR (processor)

PROCESSOR

(1) PROCESSOR (processor)

[Description Format]

```
[ Δ ] $ [ Δ ] PROCESSOR [ Δ ]( [ Δ ] processor type [ Δ ] )
[ Δ ] $ [ Δ ] PC [ Δ ] ( [ Δ ] processor type [ Δ ] )   ; Abbreviated format
```

[Function]

- The PROCESSOR control instruction specifies in a source module file the processor type of the target device subject to assembly.

[Use]

- The processor type of the target device subject to assembly must always be specified in the source module file or in the start-up command line of the assembler.
- If you omit the processor type specification for the target device subject to assembly in each source module file, you must specify the processor type at each assembly operation. Therefore, by specifying the target device subject to assembly in each source module file, you can save time and trouble when starting up the assembler.

[Explanation]

- The PROCESSOR control instruction can be described only in the header section of a source module file. If the control instruction is described elsewhere, the assembler will be aborted.
- If the specified processor type differs from the actual target device subject to assembly, the assembler will be aborted.
- Only one PROCESSOR control instruction can be specified in the module header.
- The processor type of the target device subject to assembly may also be specified with the assembler option (-c) in the start-up command line of the assembler. If the specified processor type differs between the source module file and the start-up command line, the assembler will output a warning message and give precedence to the processor type specification in the start-up command line.
- Even when the assembler option (-c) has been specified in the start-up command line, the assembler performs a syntax check on the PROCESSOR control instruction.
- If the processor type is not specified in either the source module file or the start-up command line, the assembler will be aborted.

[Application Example]

```
$       PROCESSOR ( 9026 )
$       DEBUG
$       XREF

        NAME    TEST
            :
        CSEG
```

4.3 Debug Information Output Control Instructions

The debug information output control instructions are used to specify in a source module file the output or non-output of debugging information to an object module file created from the source module file.

The following debug information output control instructions are available :

- DEBUG / NODEBUG (debug / nodebug)
- DEBUGA / NODEBUGA (debuga / nodebuga)

DEBUG / NODEBUG

(1) DEBUG / NODEBUG (debug / nodebug)

[Description Format]

```
[ Δ ] $ [ Δ ] DEBUG     ; Default assumption
[ Δ ] $ [ Δ ] DG        ; Abbreviated format
[ Δ ] $ [ Δ ] NODEBUG
[ Δ ] $ [ Δ ] NODG      ; Abbreviated format
```

[Function]

- The DEBUG control instruction tells the assembler to add local symbol information to an object module file.
- The NODEBUG control instruction tells the assembler not to add local symbol information to an object module file. However, in this case as well, the segment name is output to an object module file.
- "Local symbol information" refers to symbols other than module names and PUBLIC, EXTRN, and EXTBIT symbols.

[Use]

- Use the DEBUG control instruction when symbolic debugging including local symbols is to be performed.
- Use the NODEBUG control instruction when :
  (1) Symbolic debugging is to be performed for global symbols only
  (2) Debugging is to be performed without symbols
  (3) Only objects are required (as for evaluation with PROM)

[Explanation]

- The DEBUG or NODEBUG control instruction can be described only in the header section of a source module file.
- If the DEBUG or NODEBUG control instruction is omitted, the assembler will assume that the DEBUG control instruction has been specified.
- The addition of local symbol information can be specified using the assembler option (-g / -ng) in the start-up command line.
- If the control instruction specification in the source module file differs from the specification in the start-up command line, the specification in the command line takes precedence.
- Even when the assembler option (-ng) has been specified, the assembler performs a syntax check on the DEBUG or NODEBUG control instruction.

DEBUGA / NODEBUGA

(2) DEBUGA / NODEBUGA (debuga / nodebuga)

[Description Format]

```
[ Δ ] $ [ Δ ] DEBUGA            ; Default assumption
[ Δ ] $ [ Δ ] NODEBUGA
```

[Function]

- The DEBUGA control instruction tells the assembler to add assembler source debugging information to an object module file.
- The NODEBUGA control instruction tells the assembler not to add assembler source debugging information to an object module file.

[Use]

- Use the DEBUGA control instruction when debugging is to be performed at the assembler or structured assembler preprocessor source level. An integrated debugger will be necessary for debugging at the source level.
- Use the NODEBUGA control instruction when :
  (1) Debugging is to be performed without the assembler source
  (2) Only objects are required (as for evaluation with PROM)

[Explanation]

- The DEBUGA or NODEBUGA control instruction can be described only in the header section of a source module file.
- If the DEBUGA or NODEBUGA control instruction is omitted, the assembler will assume that the DEBUGA control instruction has been specified.
- If two or more of these control instructions are specified, the last specified control instruction takes precedence over the others.
- The addition of assembler source debugging information can be specified using the assembler option (-ga / -nga) in the start-up command line.
- If the control instruction specification in the source module file differs from the specification in the start-up command line, the specification in the command line takes precedence.
- Even when the assembler option (-nga) has been specified, the assembler performs a syntax check on the DEBUGA or NODEBUGA control instruction.
- If compiling or structure-assembling the debug information output by the C compiler or structured assembler preprocessor, do not describe the debug information output control instructions when assembling the output assemble source. The control instructions necessary at assembly are output to assembler source as control statements by the C compiler or structured assembler preprocessor.

4.4 Cross-Reference List Output Specification Control Instructions

The cross-reference list output specification control instructions are used in a source module file to specify the output or non-output of a cross-reference list.

The following cross-reference list output specification control instructions are available :

- XREF / NOXREF (xref / noxref)
- SYMLIST / NOSYMLIST (symlist / nosymlist)

XREF / NOXREF

(1) XREF / NOXREF (xref / noxref)

[Description Format]

```
[ Δ ] $ [ Δ ] XREF
[ Δ ] $ [ Δ ] XR        ; Abbreviated format
[ Δ ] $ [ Δ ] NOXREF    ; Default assumption
[ Δ ] $ [ Δ ] NOXR      ; Abbreviated format
```

[Function]

- The XREF control instruction tells the assembler to output a cross-reference list to an assembly list file.
- The NOXREF control instruction tells the assembler not to output a cross-reference list to an assembly list file.

[Use]

- Use the XREF control instruction to output a cross-reference list when you want information on where each of the symbols defined in the source module file is referenced or how many such symbols are referenced in the source module file.
- If you must specify the output or non-output of a cross-reference list at each assembly operation, you may save time and labor by specifying the XREF and NOXREF control instruction in the source module file.

[Explanation]

- The XREF or NOXREF control instruction can be described only in the header section of a source module file.
- If two or more of these control instructions are specified, the last specified control instruction takes precedence over the others.
- Output or non-output of a cross-reference list can also be specified by the assembler option (-kx / -nkx) in the start-up command line.
- If the control instruction specification in the source module file differs from the assembler option specification in the start-up command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the start-up command line, the assembler performs a syntax check on the XREF / NOXREF control instruction.

SYMLIST / NOSYMLIST

(2) SYMLIST / NOSYMLIST (symlist / nosymlist)

[Description Format]

```
[ Δ ] $ [ Δ ] SYMLIST
[ Δ ] $ [ Δ ] NOSYMLIST         ; Default assumption
```

[Function]

- The SYMLIST control instruction tells the assembler to output a symbol list to a list file.
- The NOSYMLIST control instruction tells the assembler not to output a symbol list to a list file.

[Use]

- Use the SYMLIST control instruction to output a symbol list.

[Explanation]

- The SYMLIST or NOSYMLIST control instruction can be described only in the header section of a source module file.
- If two or more of these control instructions are specified, the last specified control instruction takes precedence over the others.
- Output of a symbol list can also be specified by the assembler option (-ks / -nks) in the start-up command line.
- If the control instruction specification in the source module file differs from the assembler option specification in the start-up command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the start-up command line, the assembler performs a syntax check on the SYMLIST / NOSYMLIST control instruction.

4.5 Inclusion Control Instruction

The inclusion control instruction is used in a source module file to specify the inclusion of another module file in the source module file.

By making effective use of this control instruction, you can save time and labor in describing a source program.

The following inclusion control instruction is available :

- INCLUDE (include)

INCLUDE

(1) INCLUDE (include)

[Description Format]

```
[ Δ ] $ [ Δ ] INCLUDE [ Δ ] ( [ Δ ] filename [ Δ ] )
[ Δ ] $ [ Δ ] IC [ Δ ] ( [ Δ ] filename [ Δ ] )        ; Abbreviated format
```

[Function]

- The INCLUDE control instruction tells the assembler to insert and expand the contents of a specified file beginning on a specified line in the source program for assembly.

[Use]

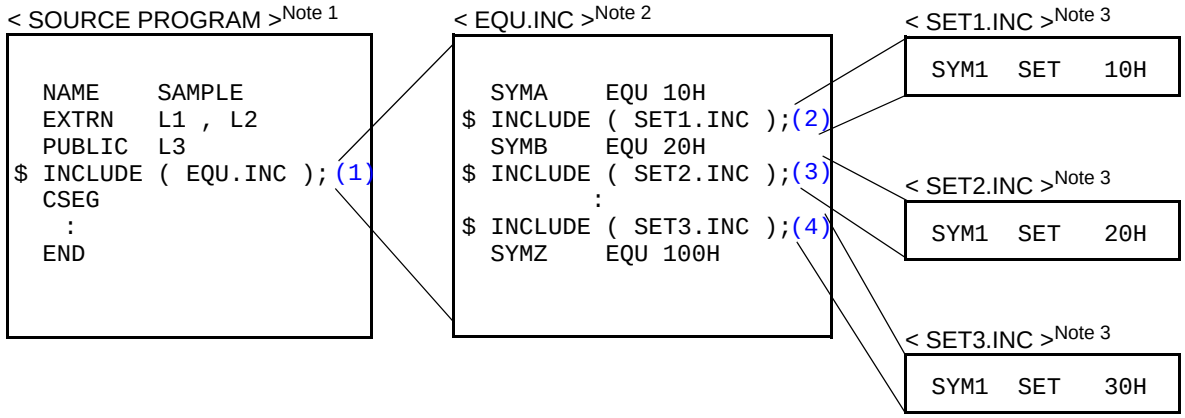
- A relatively large group of statements that may be shared by two or more source modules should be combined into a single file as an INCLUDE file. If the group of statements must be used in each source module, specify the filename of the required INCLUDE file with the INCLUDE control instruction. With this control instruction, you can greatly reduce time and labor in describing source modules.

[Explanation]

- The INCLUDE control instruction can only be described in ordinary source programs.
- The pathname or drive name of an INCLUDE file can be specified with the assembler option (-I).
- The assembler searches INCLUDE file read paths in the following sequence :
  (1) When an INCLUDE file is specified without pathname specification
    (a) Path in which the source file exists
    (b) Path specified by the assembler option (-I)
    (c) Path specified by the environment variable INC78K0S
  (2) When an INCLUDE file is specified with a pathname
    If the INCLUDE file is specified with a drive name or a pathname which begins with backslash (\ ), the path specified with the INCLUDE file will be prefixed to the INCLUDE filename. If the INCLUDE file is specified with a relative path (which does not begin with \ ), a pathname will be prefixed to the INCLUDE filename in the order described in (a) above.
- Nesting of INCLUDE files is allowed up to seven levels. In other words, the nesting level display of INCLUDE files in the assembly list is up to 8 (the term "nesting" here refers to the specification of one or more other INCLUDE files in an INCLUDE file).
- The END directive need not be described in an INCLUDE file.
- If the specified INCLUDE file cannot be opened, the assembler will abort operation.
- An INCLUDE file must be closed with IF or _IF control instruction that is properly paired with an ENDIF control instruction within the INCLUDE file. If the IF level at the entry of the INCLUDE file expansion does not correspond with the IF level immediately after the INCLUDE file expansion, the assembler will output an error message and force the IF level to return to that level at the entry of the INCLUDE file expansion.

- When defining a macro in an INCLUDE file, the macro definition must be closed in the INCLUDE file. If an ENDM directive appears unexpectedly (without the corresponding MACRO directive) in the INCLUDE file, an error message will be output and the ENDM directive will be ignored. If an ENDM directive is missing for the MACRO directive described in the INCLUDE file, the assembler will output an error message but will process the macro definition by assuming that the corresponding ENDM directive has been described.

[Application Example]



Notes 1. Two or more $IC control instructions can be specified in the source file. The same INCLUDE file may also be specified more than once.

Notes 2. Two or more $IC control instructions may be specified for INCLUDE file "EQU.INC".

Notes 3. No $IC control instruction can be specified in any of the INCLUDE files "SET1.INC", "SET2.INC", and "SET3.INC".

< Explanation >

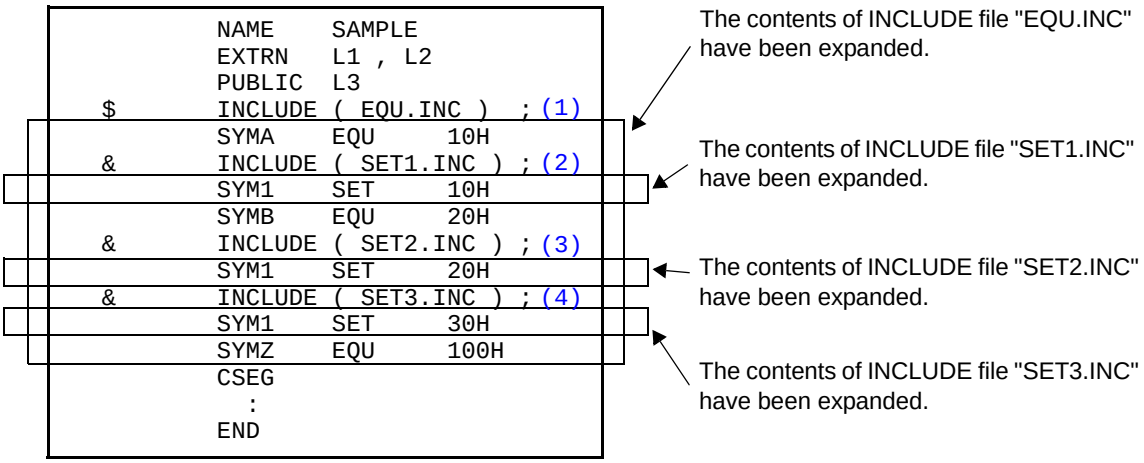
(1) This control instruction specifies "EQU.INC" as the INCLUDE file.

(2) This control instruction specifies "SET1.INC" as the INCLUDE file.

(3) This control instruction specifies "SET2.INC" as the INCLUDE file.

(4) This control instruction specifies "SET3.INC" as the INCLUDE file.

When this source program is assembled, the contents of the INCLUDE file will be expanded as follows :



4.6 Assembly List Control Instructions

The assembly list control instructions are used in a source module file to control the output format of an assembly list such as page ejection, suppression of list output, and subtitle output.

The assembly list control instructions include :

- EJECT (eject)
- LIST / NOLIST (list / nolist)
- GEN / NOGEN (generate / no generate)
- COND / NOCOND (condition / no condition)
- TITLE (title)
- SUBTITLE (subtitle)
- FORMFEED / NOFORMFEED (formfeed / noformfeed)
- WIDTH (width)
- LENGTH (length)
- TAB (tab)

EJECT

(1) EJECT (eject)

[Description Format]

```
[ Δ ] $ [ Δ ] EJECT
[ Δ ] $ [ Δ ] EJ        ; Abbreviated format
```

[Default Assumption]

- EJECT control instruction is not specified.

[Function]

- The EJECT control instruction causes the assembler to execute page ejection (formfeed) of an assembly list.

[Use]

- Describe the EJECT control instruction in a line of the source module at which page ejection of the assembly list is required.

[Explanation]

- The EJECT control instruction can only be described in ordinary source programs.
- Page ejection of the assembly list is executed after the image of the EJECT control instruction itself is output.
- If the assembler option (-np) or (-llo) is specified in the start-up command line or if the assembly list output is disabled by another control instruction, the EJECT control instruction becomes invalid. See the RA78K0S Series Assembler Package Operation User's Manual for those assembler options.
- If an illegal description follows the EJECT control instruction, the assembler will output an error message.

[Application Example]

```
            :
        MOV     [ DE+ ] , A
        BR      $$
$       EJECT                   ; (1)
        CSEG
            :
        END
```

< Explanation >

(1) When page ejection is executed with the EJECT control instruction, the output assembly list will look like this.

< Assembly list >

```
            :
        MOV     [ DE+ ] , A
        BR      $$
$       EJECT                   ; (1)
------------------------------------------------ age ejection ---------
        CSEG
            :
        END
```

LIST / NOLIST

(2) LIST / NOLIST (list / nolist)

[Description Format]

```
[ Δ ] $ [ Δ ] LIST      ; Default assumption
[ Δ ] $ [ Δ ] LI        ; Abbreviated format
[ Δ ] $ [ Δ ] NOLIST
[ Δ ] $ [ Δ ] NOLI      ; Abbreviated format
```

[Function]

- The LIST control instruction indicates to the assembler the line at which assembly list output must start.
- The NOLIST control instruction indicates to the assembler the line at which assembly list output must be suppressed.

  All source statements described after the NOLIST control instruction specification will be assembled, but will not be output on the assembly list until the LIST control instruction appears in the source program.

[Use]

- Use the NOLIST control instruction to limit the amount of assembly list output.
- Use the LIST control instruction to cancel the assembly list output suppression specified by the NOLIST control instruction.

  By using a combination of NOLIST and LIST control instructions, you can control the amount of assembly list output as well as the contents of the list.

[Explanation]

- The LIST / NOLIST control instruction can only be described in ordinary source programs.
- The NOLIST control instruction functions to suppress assembly list output and is not intended to stop the assembly process.
- If the LIST control instruction is specified after the NOLIST control instruction, statements described after the LIST control instruction will be output again on the assembly list. The image of the LIST or NOLIST control instruction will also be output on the assembly list.
- If neither the LIST nor NOLIST control instruction is specified, all statements in the source module will be output to an assembly list.

[Application Example]

```
        NAME    SAMP1
$       NOLIST        ; (1)
DATA1   EQU     10H   ; The statement will not be output to the assembly list.
DATA2   EQU     11H   ; The statement will not be output to the assembly list.
            :         ; The statement will not be output to the assembly list.
DATAX   EQU     20H   ; The statement will not be output to the assembly list.
DATAY   EQU     20H   ; The statement will not be output to the assembly list.
$       LIST          ; (2)
        CSEG
            :
        END
```

< Explanation >

(1) Because the NOLIST control instruction is specified here, statements after "$ NOLIST" and up to the LIST control instruction in (2) will not be output on the assembly list. The image of the NOLIST control instruction itself will be output on the assembly list.

(2) Because the LIST control instruction is specified here, statements after this control instruction will be output again on the assembly list. The image of the LIST control instruction itself will also be output on the assembly list.

GEN / NOGEN

(3) GEN / NOGEN (generate / no generate)

[Description Format]

```
[ Δ ] $ [ Δ ] GEN       ; Default assumption
[ Δ ] $ [ Δ ] NOGEN
```

[Function]

- The GEN control instruction tells the assembler to output macro definition lines, macro reference lines, and macro-expanded lines to an assembly list.
- The NOGEN control instruction tells the assembler to output macro definition lines and macro reference lines but to suppress macro-expanded lines.

[Use]

- Use the GEN / NOGEN control instruction to limit the amount of assembly list output.

[Explanation]

- The GEN / NOGEN control instruction can only be described in ordinary source programs.
- If neither the GEN nor NOGEN control instruction is specified, macro definition lines, macro reference lines, and macro-expanded lines will be output to an assembly list.
- The specified list control takes place after the image of the GEN or NOGEN control instruction itself has been printed on the assembly list.
- The assembler continues its processing and increments the statement number (STNO) count even after the list output control by the NOGEN control instruction.
- If the GEN control instruction is specified after the NOGEN control instruction, the assembler will resume the output of macro-expanded lines.

[Application Example]

```
        NAME    SAMP
$       NOGEN               ; (1)
ADMAC   MACRO   PARA1 , PARA2
                MOV     A , #PARA1
                ADD     A , #PARA2
        ENDM
        CSEG
        ADMAC   10H , 20H
        END
```

When the above source program is assembled, the output assembly list will look like this. When this source program is assembled, the assembly list will be expanded as follows :

< Assembly list >

```
        NAME    SAMP1
$       NOGEN               ; (1)
ADMAC   MACRO   PARA1 , PARA2
                MOV     A , #PARA1
                ADD     A , #PARA2
        ENDM
        CSEG
        ADMAC   10H , 20H
        MOV     A , #10H    ; The macro-expanded lines will not be output.
        AUD     A , #20H    ; The macro-expanded lines will not be output.
        END
```

< Explanation >

(1) Because the NOGEN control instruction is specified, the macro-expanded lines will not be output to the assembly list.

COND / NOCOND

(4) COND / NOCOND (condition / no condition)

[Description Format]

```
[ Δ ] $ [ Δ ] COND      ; Default assumption
[ Δ ] $ [ Δ ] NOCOND
```

[Function]

- The COND control instruction tells the assembler to output lines that have satisfied the conditional assembly condition as well as those which have not satisfied the conditional assembly condition to an assembly list.
- The NOCOND control instruction tells the assembler to output only lines that have satisfied the conditional assembly condition to an assembly list. The output of lines that have not satisfied the conditional assembly condition and lines in which IF / _IF, ELSEIF / _ELSEIF, ELSE, and ENDIF have been described will be suppressed.

[Use]

- Use the COND / NOCOND control instruction to limit the amount of assembly list output.

[Explanation]

- The COND / NOCOND control instruction can only be described in ordinary source programs.
- If neither the COND nor NOCOND control instruction is specified, the assembler will output lines that have satisfied the conditional assembly condition as well as those which have not satisfied the conditional assembly condition to an assembly list.
- The specified list control takes place after the image of the COND or NOCOND control instruction itself has been printed on the assembly list.
- The assembler increments the ALNO and STNO counts even after the list output control by the NOCOND control instruction.
- If the COND control instruction is specified after the NOCOND control instruction, the assembler will resume the output of lines that have not satisfied the conditional assembly condition and lines in which IF / _IF, ELSEIF / _ELSEIF, ELSE, and ENDIF have been described.

[Application Example]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )              ; This part , though assembled , will not
                                ; be outout to the list.
                MOV A , #1H
$       ELSE                    ; This part , though assembled , will not
                                ; be outout to the list.
                MOV A , #0H     ; This part , though assembled , will not
                                ; be outout to the list.
$       ENDIF                   ; This part , though assembled , will not
                                ; be outout to the list.
        END
```

TITLE

(5) TITLE (title)

[Description Format]

```
[ Δ ] $ [ Δ ] TITLE [ Δ ] ( [ Δ ] ' title - string ' [ Δ ] )
[ Δ ] $ [ Δ ] TT [ Δ ] ( [ Δ ] ' title - string ' [ Δ ] ) ; Abbreviated format
```

[Default Assumption]

- When the TITLE control instruction is not specified, the TITLE column of the assembly list header is left blank.

[Function]

- The TITLE control instruction specifies the character string to be printed in the TITLE column at each page header of an assembly list, symbol table list, or cross-reference list.

[Use]

- Use the TITLE control instruction to print a title on each page of a list so that the contents of the list can be easily identified.
- If you need to specify a title with the assembler option at each assembly time, you can save time and labor in starting the assembler by describing this control instruction in the source module file.

[Explanation]

- The TITLE control instruction can be described only in the header section of a source module file.
- If two or more TITLE control instructions are specified at the same time, the assembler will accept only the last specified TITLE control instruction as valid.
- Up to 60 characters can be specified as the title string. If the specified title string consists of 61 or more characters, the assembler will accept only the first 60 characters of the string as valid.
  However, if the character length specification per line of an assembly list file (a quantity "X") is 119 characters or less, "X - 60 characters" will be acceptable.
- If a single quotation mark (') is to be used as part of the title string, describe the single quotation mark twice in succession.
- If no title string is specified (the number of characters in the title string = 0), the assembler will leave the TITLE column blank.
- If any character not included in "2.2.2 Character set" is found in the specified title string, the assembler will output "!" in place of the illegal character in the TITLE column.
- A title for an assembly list can also be specified with the assembler option (-lh) in the start-up command line of the assembler.

[Application Example]

```
$       PROCESSOR ( 9026 )
$       TITLE ( ' THIS IS TITLE ' )
        NAME    SAMPLE
        CSEG
        MOV     A , B
        END
```

When the above source program is assembled, the output assembly list will appear as shown on the next (with the number of lines per page specified as 72).

< Assembly list >

```
78K/0S Series Assembler Vx.xx    THIS IS TITLE    Date:xx xxx xxxx  Page : 1

Command :       sample.asm
Para-file :
In-file :       SAMPLE.ASM
Obj-file :      SAMPLE.REL
Prn-file :      SAMPLE.PRN

         Assemble list

ALNO    STNO    ADRS    OBJECT  M I     SOURCE STATEMENT

 1       1                               $       PROCESSOR ( 9026 )
 2       2                               $       TITLE ( ' THIS IS TITLE ' )
 3       3                                       NAME    SAMPLE
 4       4       ----                            CSEG
 5       5       0000    63                      MOV     A , B
 6       6                                       END

Segment information :

ADRS    LEN     NAME

0000    0001H   ?CSEG

Target chip : uPD789026
Device file : Vx.xx
Assembly complete , 0 error ( s ) and 0 warning ( s ) found. ( 0 )
```

SUBTITLE

(6) SUBTITLE (subtitle)

[Description Format]

```
[ Δ ] $ [ Δ ] SUBTITLE [ Δ ] ( [ Δ ] ' title - string ' [ Δ ] )
[ Δ ] $ [ Δ ] ST [ Δ ] ( [ Δ ] ' title - string ' [ Δ ] ) ; Abbreviated format
```

[Default Assumption]

- When the SUBTITLE control instruction is not specified, the SUBTITLE section of the assembly list header is left blank.

[Function]

- The SUBTITLE control instruction specifies the character string to be printed in the SUBTITLE section at each page header of an assembly list.

[Use]

- Use the SUBTITLE control instruction to print a subtitle on each page of an assembly list so that the contents of the assembly list can be easily identified. The character string of a subtitle may be changed for each page.

[Explanation]

- The SUBTITLE control instruction can only be described in ordinary source programs.
- Up to 72 characters can be specified as the subtitle string.
  If the specified title string consists of 73 or more characters, the assembler will accept only the first 72 characters of the string as valid. A 2-byte character is counted as two characters, and tab is counted as one character.
- The character string specified with the SUBTITLE control instruction will be printed in the SUBTITLE section on the page after the page on which the SUBTITLE control instruction has been specified. However, if the control instruction is specified at the top (first line) of a page, the subtitle will be printed on that page.
- If the SUBTITLE control instruction has not been specified, the assembler will leave the SUBTITLE section blank.
- If a single quotation mark (') is to be used as part of the character string, describe the single quotation mark twice in succession.
- If the character string in the SUBTITLE section is 0, the SUBTITLE column will be left blank.
- If any character not included in "2.2.2 Character set" is found in the specified subtitle string, the assembler will output "!" in place of the illegal character in the SUBTITLE column. If CR (0DH) is described, an error will result and nothing will be output in the assembly list. If 00H is described, nothing from that point to the closing single quotation mark (') will be output.

[Application Example]

```
        NAME    SAMP
        CSEG
$       SUBTITLE ( ' THIS IS SUBTITLE 1 ' )     ; (1)
$       EJECT                                   ; (2)
        CSEG
$       SUBTITLE ( ' THIS IS SUBTITLE 2 ' )     ; (3)
$       EJECT                                   ; (4)
$       SUBTITLE ( ' THIS IS SUBTITLE 3 ' )     ; (5)
        END
```

< Explanation >

(1) This control instruction specifies the character string "THIS IS SUBTITLE 1".

(2) This control instruction specifies a page ejection.

(3) This control instruction specifies the character string "THIS IS SUBTITLE 2".

(4) This control instruction specifies a page ejection.

(5) This control instruction specifies the character string "THIS IS SUBTITLE 3".

The assembly list for this example appears as follows (with the number of lines per page specified as 80).

< Assembly list >

```
78K/0S Series Assembler Vx.xx Date : xx xxx xxxx  Page : 1

Command :       -c9026 sample.asm
Para-file :
In-file :       SAMPLE.ASM
Obj-file :      SAMPLE.REL
Prn-file :      SAMPLE.PRN

        Assemble list

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 1     1                        NAME SAMP
 2     2    ----                CSEG
 3     3                     $  SUBTITLE ( ' THIS IS SUBTITLE 1 ' ) ; (1)
 4     4                     $  EJECT                               ; (2)
------------------------------------------------ age ejection ---------------
78K/0S Series Assembler Vx.xx Date:xx xxx xxxx  Page: 2

THIS IS SUBTITLE 1

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 5     5    ----                CSEG
 6     6                     $  SUBTITLE ( ' THIS IS SUBTITLE 2 ' ) ; (3)
 7     7                     $  EJECT                               ; (4)
------------------------------------------------ age ejection ---------------
78K/0S Series Assembler Vx.xx Date:xx xxx xxxx  Page: 3

THIS IS SUBTITLE 2

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 8     8                     $  SUBTITLE ( ' THIS IS SUBTITLE 3 ' ) ; (5)
 9     9                        END

Target chip : uPD789026
Device file : Vx.xx
Assembly complete , 0 error ( s ) and 0 warning ( s ) found. ( 0 )
```

FORMFEED / NOFORMFEED

(7) FORMFEED / NOFORMFEED (formfeed / noformfeed)

[Description Format]

```
[ Δ ] $ [ Δ ] FORMFEED
[ Δ ] $ [ Δ ] NOFORMFEED        ; Default assumption
```

[Function]

- The FORMFEED control instruction tells the assembler to output a FORMFEED code at the end of an assembly list file.
- The NOFORMFEED control instruction tells the assembler not to output a FORMFEED code at the end of an assembly list file.

[Use]

- Use the FORMFEED control instruction when you want to start a new page after printing the contents of an assembly list file.

[Explanation]

- The FORMFEED or NOFORMFEED control instruction can be described only in the header section of a source module file.
- At the time of printing an assembly list, the last page of the list may not come out if printing ends in the middle of a page. In such a case, add a FORMFEED code to the end of the assembly list using the FORMFEED control instruction or assembler option (-lf).

  In many cases, a FORMFEED code will be output at the end of a file. For this reason, if a FORMFEED code exists at the end of a list file, an unwanted white page may be ejected. To prevent this, the NOFORMFEED control instruction or assembler option (-nlf) has been set as a default value.
- If two or more FORMFEED / NOFORMFEED control instructions are specified at the same time, only the last specified control instruction will become valid.
- The output or non-output of a formfeed code may also be specified with the assembler option (-lf) or (-nlf) in the start-up command line of the assembler.
- If the control instruction specification (FORMFEED / NOFORMFEED) in the source module differs from the specification (-lf / -nlf) in the start-up command line, the specification in the start-up command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the start-up command line, the assembler performs a syntax check on the FORMFEED or NOFORMFEED control instruction.

WIDTH

(8) WIDTH (width)

[Description Format]

```
[ Δ ] $ [ Δ ] WIDTH [ Δ ] ( [ Δ ] columns - per - line [ Δ ] )
```

[Default Assumption]

$WIDTH (132)

[Function]

- The WIDTH control instruction specifies the number of columns (characters) per line of a list file. "columns-per-line" must be a value in the range of 72 to 260.

[Use]

- Use the WIDTH control instruction when you want to change the number of columns per line of a list file.

[Explanation]

- The WIDTH control instruction can be described only in the header section of a source module file.
- If two or more WIDTH control instructions are specified at the same time, only the last specified control instruction will become valid.
- The number of columns per line of a list file may also be specified with the assembler option (-lw) in the start-up command line of the assembler.
- If the control instruction specification (WIDTH) in the source module differs from the specification (-lw) in the start-up command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the start-up command line, the assembler performs a syntax check on the WIDTH control instruction.

LENGTH

(9) LENGTH (length)

[Description Format]

```
[ Δ ] $ [ Δ ] LENGTH [ Δ ] ( [ Δ ] lines - per - page [ Δ ] )
```

[Default Assumption]

$LENGTH (66)

[Function]

- The LENGTH control instruction specifies the number of lines per page of a list file. "lines-per-page" may be "0" or a value in the range of 20 to 32767.

[Use]

- Use the LENGTH control instruction when you want to change the number of lines per page of a list file.

[Explanation]

- The LENGTH control instruction can be described only in the header section of a source module file.
- If two or more LENGTH control instructions are specified at the same time, only the last specified control instruction will become valid.
- The number of columns per line of a list file may also be specified with the assembler option (-ll) in the start-up command line of the assembler.
- If the control instruction specification (LENGTH) in the source module differs from the specification (-ll) in the start-up command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the start-up command line, the assembler performs a syntax check on the LENGTH control instruction.

TAB

(10) TAB (tab)

[Description Format]

```
[ Δ ] $ [ Δ ] TAB [ Δ ] ( [ Δ ] number - of - columns [ Δ ] )
```

[Default Assumption]

$TAB (8)

[Function]

- The TAB control instruction specifies the number of columns as tab stops on a list file. "number-of-columns" may be a value in the range of 0 to 8.
- The TAB control instruction specifies the number of columns that becomes the basis of tabulation processing to output any list by replacing a HT (Horizontal Tabulation) code in a source module with several blank characters on the list.

[Use]

- Use HT code to reduce the number of blanks when the number of characters per line of any list is reduced using the TAB control instruction.

[Explanation]

- The TAB control instruction can be described only in the header section of a source module file.
- If two or more TAB control instructions are specified at the same time, only the last specified control instruction will become valid.
- The number of tab stops may also be specified with the assembler option (-lt) in the start-up command line of the assembler.
- If the control instruction specification (TAB) in the source module differs from the specification (-lt) in the start-up command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the start-up command line, the assembler performs a syntax check on the TAB control instruction.

4.7 Conditional Assembly Control Instructions

The conditional assembly control instructions are used to select a series of statements in a source module as those subject to assembly or not subject to assembly, by setting switches for conditional assembly.

These control instructions consist of the IF / _IF / ELSEIF / _ELSEIF / ELSE / ENDIF control instructions and the SET / RESET (set / reset) control instructions.

By making effective use of these control instructions, you can assemble a source module that excludes unwanted statements, with little or no change to the source module.

The following conditional assembly control instructions are available :

- IF / _IF / ELSEIF / _ELSEIF / ELSE / ENDIF
- SET / RESET (set / reset)

IF / _IF / ELSEIF / _ELSEIF / ELSE / ENDIF

(1) IF / _IF / ELSEIF / _ELSEIF / ELSE / ENDIF

[Description Format]

```
    [ Δ ] $ [ Δ ] IF [ Δ ] ( [ Δ ] switch - name [ [ Δ ] : [ Δ ] switch - name
] ... [ Δ ] )
or  [ Δ ] $ [ Δ ] _IFDconditional - expression
          :
[ Δ ] $ [ Δ ] ELSEIF [ Δ ] ( [ Δ ] switch - name [ [ Δ ] : [ Δ ] switch - name
] ... [ Δ ] )
or  [ Δ ] $ [ Δ ] _ELSEIFDconditional - expression
          :
[ Δ ] $ [ Δ ] ELSE
          :
[ Δ ] $ [ Δ ] ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.

  Source statements described between the IF or _IF control instruction and the ENDIF control instruction are subject to conditional assembly.

- If the evaluated value of the conditional expression or the switch name specified by the IF or _IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF / _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.

  If the IF or _IF condition is false (00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF / _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.

- The ELSEIF or _ELSEIF control instruction is checked for true / false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).

  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF / _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.

  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF / _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.

- If the conditions of all the IF / _IF and ELSEIF / _ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.

- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Explanation]

- The IF and ELSEIF control instructions are used for true / false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true / false condition judgment with a conditional expression.
  Both IF / ELSEIF and _IF / _ELSEIF may be used in combination. In other words, ELSEIF / _ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "2.2.3 Symbol field"). However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:). Up to five switch names can be used per module.
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET / RESET (set / reset) control instruction. Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error will result.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application Examples]

< Example 1 >

```
        text0
$       IF ( SW1 )              ; (1)
                text1
$       ENDIF                   ; (2)
            :
        END
```

< Explanation >

(1) If the value of switch name "SW1" is true, statements in "text1" will be assembled.

If the value of switch name "SW1" is false, statements in "text1" will not be assembled.

The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".

(2) This instruction indicates the end of the source statement range for conditional assembly.

< Example 2 >

```
        text0
$       IF ( SW1 )              ; (1)
                text1
$       ELSE                    ; (2)
                text2
$       ENDIF                   ; (3)
            :
        END
```

< Explanation >

(1) The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".

If the value of switch name "SW1" is true, statements in "text1" will be assembled and statements in "text2" will not be assembled.

(2) If the value of switch name "SW1" in (1) is false, statements in "text1" will not be assembled and statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

< Example 3 >

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
            :
        END
```

< Explanation >

(1) The values of switch names "SW1", "SW2", and "SW3" have been set to true or false with the SET or RESET control instruction described in "text0".

If the value of switch name "SW1" or "SW2" is true, statements in "text1" will be assembled and statements in "text2", "text3", and "text4" will not be assembled.

If the values of switch names "SW1" and "SW2" are false, statements in "text1" will not be assembled and statements after (2) will be conditionally assembled.

(2) If the values of switch names "SW1" and "SW2" in (1) are false and the value of switch name "SW3" is true, statements in "text2" will be assembled and statements in "text1", "text3", and "text4" will not be assembled.

(3) If the values of switch names "SW1" and "SW2" in (1) and "SW3" in (2) are false and the value of switch name "SW4" is true, statements in "text3" will be assembled and statements in "text1", "text2", and "text4" will not be assembled.

(4) If the values of switch names "SW1" and "SW2" in (1), "SW3" in (2), and "SW4" in (3) are all false, statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be assembled.

(5) This instruction indicates the end of the source statement range for conditional assembly.

< Example 4 >

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
            :
        END
```

< Explanation >

(1) The value of switch name "SYMA" has been defined with the EQU or SET directive described in "text0". If the symbol name "SYMA" is true (not "0"), statements in "text1" will be assembled and "text2" will not be assembled.

(2) If the value of the symbol name "SYMA" is "0", and "SYMB" and "SYMC" have the same value, statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

SET / RESET

(2) SET / RESET (set / reset)

[Description Format]

```
[ Δ ] $ [ Δ ] SET [ Δ ] ( [ Δ ] switch - name [ [ Δ ] : [ Δ ] switch - name ]
... [ Δ ] )
[ Δ ] $ [ Δ ] RESET [ Δ ] ( [ Δ ] switch - name [ [ Δ ] : [ Δ ] switch - name
] ... [ Δ ] )
```

[Function]

- The SET and RESET control instructions give a value to each switch name to be specified with the IF or ELSEIF control instruction.
- The SET control instruction gives a true value (0FFH) to each switch name specified in its operand.
- The RESET control instruction gives a false value (00H) to each switch name specified in its operand.

[Use]

- Describe the SET control instruction to give a true value (0FFH) to each switch name to be specified with the IF or ELSEIF control instruction.
- Describe the RESET control instruction to give a false value (00H) to each switch name to be specified with the IF or ELSEIF control instruction.

[Explanation]

- With the SET and RESET control instructions, at least one switch name must be described.
  The conventions for describing switch names are the same as the conventions for describing symbols (see "2.2.3 Symbol field"). However, the maximum number of characters that can be recognized as a switch name is always 31.
- The specified switch name(s) may be the same as user-defined symbol(s) other than reserved words and other switch names.
- If two or more switch names are to be specified with the SET or RESET control instruction, delimit each switch name with a colon (:). Up to 1,000 switch names can be used per module.
- A switch name once set to "true" with the SET control instruction can be changed to "false" with the RESET control instruction, and vice versa.
- A switch name to be specified with the IF or ELSEIF control instruction must be defined at least once with the SET or RESET control instruction in the source module before describing the IF or ELSEIF control instruction.
- Switch names will not be output to a cross-reference list.

[Application Example]

```
$       SET ( SW1 )             ; (1)
            :
$       IF ( SW1 )              ; (2)
                text1
$       ENDIF                   ; (3)
            :
$       RESET ( SW1 : SW2 )     ; (4)
            :
$       IF ( SW1 )              ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE                    ; (7)
                text4
$       ENDIF                   ; (8)
            :
        END
```

< Explanation >

(1) This instruction gives a true value (0FFH) to switch name "SW1".

(2) Because the true value has been given to switch name "SW1" in (1) above, statements in "text1" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly that starts from (2).

(4) This instruction gives a false value (00H) to switch names "SW1" and "SW2", respectively.

(5) Because the false value has been given to switch name "SW1" in (4) above, statements in "text2" will not be assembled.

(6) Because the false value has also been given to switch name "SW2" in (4) above, statements in "text3" will not be assembled.

(7) Because both switch names "SW1" and "SW2" are false in (5) and (6) above, statements in "text4" will be assembled.

(8) This instruction indicates the end of the source statement range for conditional assembly that starts from (5).

4.8 Other Control Instructions

The following control instructions are special control instructions output by high-level programs such as C compiler and structured assembler preprocessor :

- $TOL_INF
- $DGS
- $DGL

CHAPTER 5 MACROS

This chapter explains how to use a macro function. A macro is a very useful function when you need to describe a series of statements repeatedly in a source program.

5.1 Overview

When you must describe a series or group of instructions repeatedly in a source program, a macro function is very useful for program description. The macro function refers to the expansion of a series of statements (an instruction group) defined as a macro body with MACRO and ENDM directives into the location where the macro name is referenced.

A macro is used to increase the coding efficiency of a source program and is different from a subroutine.

Macros and subroutines have distinct features as explained below. For effective use, select either a macro or a subroutine according to the specific purpose.

(1) Subroutines

- Describe a process that must be repeated many times in a program as a single subroutine. The subroutine will be converted into machine language by the assembler only once.
- To call the subroutine, you only need to describe a subroutine call instruction (generally, instructions to set arguments are also described before and after the subroutine).
  Effective use of subroutines enables program memory to be used with high efficiency.
- By coding a series of processes in a program as subroutines, the program can be structured (this structuring makes the overall structure of the program easy for the programmer to understand, making program design easy).

(2) Macros

- The basic function of a macro is the replacement of a group of instructions with a name.
  A series (or group) of instructions defined as a macro body with MACRO and ENDM directives will be expanded into the location where the macro name is referenced.
- When the assembler finds a macro reference, the assembler expands the macro body and converts the group of instructions into machine language while replacing the formal parameter(s) of the macro body with the actual parameters at the time of the macro reference.
- Parameters can be described for a macro.
  For example, if there are instruction groups that are the same in processing procedure but are different in the data to be described in the operand, define a macro by assigning formal parameter(s) to the data. By describing the macro name and the actual parameter(s) at macro reference time, the assembler can cope with various instruction groups that differ only in part of the statement description.

Programming techniques using subroutines are mainly used to reduce memory size and structure programs, whereas macros are used to increase the coding efficiency of the program.

5.2 Utilization of Macros

5.2.1 Macro definition

A macro is defined with the MACRO and ENDM directives.

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
macro - name	MACRO : ENDM	[[formal - parameter [, ...]]	[; comment] [; comment]

[Function]

- The MACRO directive executes a macro definition by assigning the macro name specified in the symbol field to a series of statements (called a macro body) described between this directive and the ENDM directive.

[Application Example]

```
ADMAC   MACRO   PARA1 , PARA2
        MOV     A , #PARA1
        ADD     A , #PARA2
        ENDM
```

< Explanation >

The above example shows a simple macro definition that specifies the addition of two values "PARA1" and "PARA2" and the storage of the result in register A. The macro is given a name "ADMAC" and "PARA1" and "PARA2" are formal parameters.

For details, see "3.8 Macro Directives".

5.2.2 Macro reference

To call a macro, the already defined macro name must be described in the mnemonic field of the source program.

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[label :]	macro - name	[[actual - parameter [, ...]]	[; comment]

[Function]

- This statement description calls the macro body assigned to the macro name specified in the mnemonic field.

[Use]

- Use this statement description to call a macro body.

[Explanation]

- The macro name to be specified in the mnemonic field must have been defined before the macro reference.
- Up to 16 actual parameters may be specified per line by delimiting each actual parameter with a comma (,).
- No blank can be described in the character string constituting an actual parameter.
- When describing a comma (,), semicolon (;), blank, or tab in an actual parameter, enclose the character string that includes any of these special characters with a pair of single quotation marks.
- Formal parameters are replaced with their corresponding actual parameters in sequence from left to right.
  A warning message will be output if the number of formal parameters is not equal to the number of actual parameters.

[Application Example]

```
        NAME    SAMPLE
ADMAC   MACRO   PARA1 , PARA2
                MOV     A , #PARA1
                ADD     A , #PARA2
        ENDM

        CSEG
            :
        ADMAC   10H , 20H
            :
        END
```

< Explanation >

This macro reference calls the already defined macro name "ADMAC". 10H and 20H are actual parameters.

5.2.3 Macro expansion

The assembler processes a macro as follows :

- The assembler expands the macro body corresponding to the referenced macro name to the location where the macro name is referenced.
- The assembler assembles statements in the expanded macro body in the same way as other statements.

5.2.4 Application example

When the macro referenced in "5.2.2 Macro reference" is assembled, the macro body will be expanded as shown below.

```
        NAME    SAMPLE

        ; Macro definition
ADMAC   MACRO   PARA1 , PARA2
                MOV     A , #PARA1
                ADD     A , #PARA2
        ENDM

        ; Source text
        CSEG
            :

        ; Macro expansion
        ADMAC   10H , 20H               ; (1)
        MOV     A , #10H
        ADD     A , #20H

        ; Source text
            :
        END
```

< Explanation >

(1) By the macro reference in (1), the macro body will be expanded. The formal parameters within the macro body will be replaced with the actual parameters.

5.3 Symbols Within Macros

Symbols that can be defined in a macro are divided into two types : global symbols and local symbols.

(1) Global symbols

- A global symbol is a symbol that can be referenced from any statement within a source program. Therefore . if a macro in which the global symbol has been defined is referenced more than once to expand a series of statements . the symbol will cause a double definition error.
- Symbols not defined with the LOCAL directive are global symbols.

(2) Local symbols

- A local symbol is a symbol defined with the LOCAL directive (see "3.8 Macro Directives").
- A local symbol can be referenced within the macro declared as LOCAL with the LOCAL directive.
- No local symbol can be referenced from outside the macro.

[Application Example]

< Source program >

```
        NAME    SAMPLE
            ; Macro definition
MAC1    MACRO
                LOCAL   LLAB    ; (1)
LLAB :
            :
GLAB :
                BR      LLAB    ; (2)
                BR      GLAB    ; (3)
        ENDM
            :
            ; Source text
REF1 :  MAC1                    ; (4) <-- Macro reference
            :
        BR      LLAB            ; (5) <-- This description is erroneous.
        BR      GLAB            ; (6)
            :
REF2 :  MAC1                    ; (7) <-- Macro reference
            :
        END
```

< Explanation >

(1) This LOCAL directive defines label "LLAB" as a local symbol.

(2) This BR instruction references local symbol "LLAB" in macro "MAC1".

(3) This BR instruction references global symbol "GLAB" in macro "MAC1".

(4) This statement references macro "MAC1".

(5) This BR instruction references local symbol "LLAB" from outside the definition of macro "MAC1". This description causes an error when the source program is assembled.

(6) This BR instruction references global symbol "GLAB" from outside the definition of macro "MAC1".

(7) This statement references macro "MAC1". The same macro is referenced twice.

When the source program in the above example is assembled . the macro body will be expanded as shown below.

< Assembly list >

```
        NAME    SAMPLE
            :
REF1 :  MAC1
            ; Macro expansion
??RA0000 :
            :
GLAB :                          <-- Error
        BR      ??RA0000
        BR      GLAB
            ; Source text
            :
        BR      !LLAB           <-- Error
        BR      !GLAB
            :
REF2 :  MAC1
            ; Macro expansion
??RA0001 :
            :
GLAB :                          <-- Error
        BR      ??RA0001
        BR      GLAB
            ; Source text
            :
        END
```

< Explanation >

Global symbol "GLAB" has been defined in macro "MAC1". Because macro "MAC1" is referenced twice . global symbol "GLAB" causes a double definition error as a result of expanding a series of statements in the macro body.

5.4 Macro Operators

Two types of macro operators are available : "& (ampersand)" and "' (single quotation mark)".

(1) & (Concatenation)

- The ampersand "&" concatenates one character string to another within a macro body. At macro expansion time, the character string on the left of the ampersand is concatenated to the character string on the right of the sign. The "&" itself disappears after concatenating the strings.
- At macro definition time, a string before or after "&" in a symbol can be recognized as a formal parameter or LOCAL symbol. At macro expansion time, the formal parameter or LOCAL symbol before or after "&" is evaluated as a symbol and can be concatenated in the symbol.
- The "&" sign enclosed in a pair of single quotation marks is simply handled as data.
- Two "&" signs described in succession are handled as a single "&" sign.

[Application Example]

< Macro definition >

```
MAC     MACRO   P
LAB&P :                         <-- Formal parameter 'P' is recognized.
                D&B     10H
                DB      ' P '
                DB      P
                DB      ' &P '
        ENDM
```

< Macro reference >

```
        MAC     1H
LAB1H :
        DB      10H     <-- 'D' and 'B' are concatenated and become 'DB'.
        DB      ' P '
        DB      1H
        DB      ' &P '  <-- & enclosed in a pair of single quotation marks
                            is simply handled as data.
```

(2) ' (Single quotation mark)

- If a character string enclosed by a pair of single quotation marks is described at the beginning of an actual parameter in a macro reference line or an IRP directive or after a delimiting character, the character string will be interpreted as an actual parameter. The character string will be passed to the actual parameter without the enclosing single quotation marks.
- If a character string enclosed by a pair of single quotation marks exists in a macro body, the character string will simply be handled as data.
- To use a single quotation mark as a single quotation mark in text, describe the single quotation mark twice in succession.

[Application Example]

```
        NAME    SAMP
MAC1    MACRO   P
                IRP     Q , < P >
                        MOV     A , #Q
                ENDM
        ENDM

        MAC1    ' 10 , 20 , 30 '
```

When the source program in the above example is assembled, macro "MAC1" will be expanded as shown below.

< Assembly list >

```
IRP     Z , < 10 , 20 , 30 >
        MOV     A , #Q
ENDM
MOV     A , #10                 ; IRP expansion
MOV     A , #20                 ; IRP expansion
MOV     A , #30                 ; IRP expansion
```

CHAPTER 6 PRODUCT UTILIZATION

This chapter introduces some measures recommended for effective utilization of the RA78K0S assembler package.

6.1 Saving Time and Trouble in Starting Up the Assembler

Some control instructions have the same functions as assembler options and must always be used when starting up the assembler; examples of these include the processor type specification (-c). It is advisable to describe such control instructions in a source module file. In particular, the processor type specification, which cannot be omitted, should be specified in the header section of a source module file using the PROCESSOR control instruction. This avoids the need to specify the assembler option (-c) in the start-up command line each time you start up the assembler program. Remember that an error will result if you forget to specify this assembler option in the start-up command line, and you will have to start up the assembler from the beginning again with the correct assembler options.

The cross-reference list output control instruction (XREF) should also be specified in the module header.

< Example >

```
$       PROCESSOR ( 9026 )
$       KANJICODE SJIS
$       XRFF

        NAME    TEST

        CSEG
            :
        END
```

6.2 How to Develop Programs with High Memory Utilization Efficiency

The short direct addressing area is an area that can be accessed with instructions of short byte length as compared with other data memory areas.

Therefore, by using this area efficiently, a program with high memory utilization efficiency can be developed.

Declare the short direct addressing area in one module. In this way, even if all the variables which you intended to locate in the short direct addressing area cannot be located there, you can make changes easily so that only variables to be accessed frequently are located in the short direct addressing area.

[Application Example]

< Module 1 >

```
        PUBLIC  TMP1 , TMP2
WORK    DSEG    AT      0FE20H
TMP1 :  DS      2               ; word
TMP2 :  DS      1               ; byte
```

< Module 2 >

```
        EXTRN   TMP1 , TMP2
SAB     CSEG
        MOVW    TMP1 , #1234H
        MOV     TMP2 , #56H
            :
```

APPENDIX A LIST OF RESERVED WORDS

Reserved words are available in six types : machine language instructions, directives, control instructions, operators, register names, and sfr symbols. The reserved words are character strings reserved in advance by the assembler and cannot be used for other than the intended purposes.

Types of reserved words that can be described in the respective fields of a source program are shown below.

Table A-1 Types of Reserved Words

Type	Explanation
Symbol field	No reserved words can be described in this field.
Mnemonic field	Only machine language instructions and directives can be described in this field.
Operand field	Only operators, sfr symbols, and register names can be described in this field.
Comment field	All reserved words can be described in this field.

For the sfr list, refer to the user's manual of each device.

For the interrupt request source list, refer to the user's manual of each device.

For the machine language instructions and list of register names, refer to the user's manual of each device.

Table A-2 List of Reserved Words

Type	Reserved Word				
Operators	AND	BITPOS	DATAPOS	EQ (or =)	GE (or >=)
	GT (or >)	HIGH	LE (or <=)	LOW	LT (or <)
	MASK	MOD	NE (or < >)	NOT	OR
	SHL	SHR	XOR		
Directives	AT	BR	BSEG	CALLT0	CSEG
	DB	DBIT	DS	DSEG	DSPRAM
	DW	END	ENDM	EQU	EXITM
	EXTBIT	EXTRN	FIXED	IHRAM	IRP
	IXRAM	LOCAL	LRAM	MACRO	NAME
	ORG	PUBLIC	REPT	SADDR	SADDRP
	SET	UNIT	UNITP		
Control instructions	COND / NOCOND			DEBUG / NODEBUG	
	DEBUGA / NODEBUGA [DG / NODG]			EJECT [EJ]	
	FORMFEED / NOFORMFEED			GEN / NOGEN	
	IF / _IF / ELSEIF / _ELSEIF / ELSE / ENDIF			INCLUDE [IC]	
	LENGTH			LIST / NOLIST [LI / NOLI]	
	PROCESSOR [PC]			SET / RESET	
	SUBTITLE [ST]			SYMLIST / NOSYMLIST	
	TAB			TITLE [TT]	
	WIDTH			XREF / NOXREF [XR / NOXR]	
Others	DGL	DGS	SFR	SFRP	TOL_INF

Remark The items in brackets following the control instructions indicate the abbreviated format.

APPENDIX B LIST OF DIRECTIVES

Table B-1 List of directives

Directive				Function Classification	Remarks
Symbol Field	Mnemonic Field	Operand Field	Comment Field		
[segment name]	CSEG	[relocation - attribute]	[; comment]	Declares the start of a code segment.	
[segment name]	DSEG	[relocation - attribute]	[; comment]	Declares the start of a data segment.	
[segment name]	BSEG	[relocation - attribute]	[; comment]	Declares the start of a bit segment.	
[segment name]	ORG	absolute - expression	[; comment]	Declares the start of an absolute segment.	Forward reference of symbols within an operand is prohibited.
name	EQU	expression	[; comment]	Defines a name.	name : symbol Forward or external reference of symbols within an operand is prohibited.
name	SET	absolute - expression	[; comment]	Defines a redefinable name.	name : symbol Forward reference of symbols within an operand is prohibited.
[label :]	DB	(size) initial value [, ...]	[; comment]	Initializes or reserves a byte data area.	label : symbol A character string can be located in place of an initial value.
[label :]	DW	(size) initial value [, ...]	[; comment]	Initializes or reserves a word data area.	label : symbol
[label :]	DS	absolute - expression	[; comment]	Reserves byte data area.	name : symbol Forward reference of symbols within an operand is prohibited.
name	DBIT	None	[; comment]	Reserves a bit data area.	name : symbol Forward reference of symbols within an operand is prohibited.

Table B-1 List of directives

Directive				Function Classification	Remarks
Symbol Field	Mnemonic Field	Operand Field	Comment Field		
[label :]	PUBLIC	symbol - name [, ...]	[; comment]	Declares an external definition name.	
[label :]	EXTRN	symbol - name [, ...]	[; comment]	Declares an external reference name.	
[label :]	EXTBIT	bit - symbol - name [, ...]	[; comment]	Declares an external reference name.	Symbol names are limited to those having a bit value.
[label :]	NAME	object - module - name	[; comment]	Defines a module name.	module name : symbol
[label :]	BR	expression	[; comment]	Automatically selects a branch instruction.	label : symbol
macro - name	MACRO	[formal - parameter [, ...]]	[; comment]	Defines a macro.	macro - name : symbol
[label :]	LOCAL	symbol - name [, ...]	[; comment]	Defines a symbol valid only within a macro.	Can only be used in the macro definition.
[label :]	REPT	absolute - expression	[; comment]	Specifies repeat count during macro expansion.	label : symbol
[label :]	IRP	formal - parameter, <actual - parameter [, ...]>	[; comment]	Assigns an actual parameter to a formal parameter.	label : symbol
[label :]	EXITM	None	[; comment]	Interrupts macro expansion.	Can only be used in the macro definition.
None	ENDM	None	[; comment]	Terminates macro definition.	Can only be used in the macro definition.
None	END	None	[; comment]	Indicates the end of the source module.	

APPENDIX C INDEX

W

X

For further information, please contact:

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668,
Japan
Tel: 044-435-5111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554, U.S.A.
Tel: 408-588-6000
800-366-9782
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211-65030
http://www.eu.necel.com/

Hanover Office
Podbielskistrasse 166 B
30177 Hannover
Tel: 0 511 33 40 2-0

Munich Office
Werner-Eckert-Strasse 9
81829 München
Tel: 0 89 92 10 03-0

Stuttgart Office
Industriestrasse 3
70565 Stuttgart
Tel: 0 711 99 01 0-0

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908-691-133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01-3067-5800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091-504-2787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 638 72 00

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02-667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven
The Netherlands
Tel: 040 265 40 10

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27 ZhiChunLu Haidian
District, Beijing 100083, P.R.China
Tel: 010-8235-1155
http://www.cn.necel.com/

Shanghai Branch
Room 2509-2510, Bank of China Tower,
200 Yincheng Road Central,
Pudong New Area, Shanghai, P.R.China P.C:200120
Tel:021-5888-5400
http://www.cn.necel.com/

Shenzhen Branch
Unit 01, 39/F, Excellence Times Square Building,
No. 4068 Yi Tian Road, Futian District, Shenzhen,
P.R.China P.C:518048
Tel:0755-8282-9800
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
Unit 1601-1613, 16/F., Tower 2, Grand Century Place,
193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: 2886-9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R. O. C.
Tel: 02-8175-9600
http://www.tw.necel.com/

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253-8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku,
Seoul, 135-080, Korea
Tel: 02-558-3737
http://www.kr.necel.com/

G0706