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.

RENESAS

User's Manual

RA17K

Relocatable Assembler

Document No. U10305EJ2V0UM00 (2nd edition)
Date Published November 1997 J

[MEMO]

SUMMARY OF CONTENTS

Regional Information

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

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

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

NEC Electronics Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

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

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

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

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

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

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

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

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

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

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

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

NEC do Brasil S.A.
Cumbica-Guarulhos-SP, Brasil
Tel: 011-6465-6810
Fax: 011-6465-6829

Major Changes

Page	Description
Throughout	The version of this manual has been changed from V1.0 to V2.0.
p.6	**Section 1.2.2** The ENTRY pseudo instruction has been deleted.
p.10	**Section 1.4** has been added.
p.10	In **Section 1.5**, Windows™ has been added as an OS.
p.13	**Section 1.7** has been changed.
p.13	**Section 1.8** has been added.
p.303	**Chapter 21** The following instructions have been added: • System register operation instructions • Extended instructions • Structured instructions
p.305	Instructions have been added to **Section 21.1**.
p.306	The description has been added to **Section 21.1.1**.
p.313	The description has been added to **Section 21.2**.
p.319	The note in **Section 21.2.1** has been deleted.
p.328	Instructions have been added to **Section 21.3**.
p.390	**Section 21.4** has been added.
 p.484 p.484	**Chapter 24** Error message F147 has been added. Error message W149 has been added.

The mark ✳ shows major revised points.

[MEMO]

PREFACE

Outline of product

RA17K is the relocatable assembler for the development of 17K series software.

Intended readers

This manual is aimed at those engineers working with the 17K series 4-bit single-chip micro controller, who are responsible for designing and developing related applications by using RA17K or *emIC-17K*™.

Organization

This manual is organized as follows:

- Overview
- Source program
- Pseudo instructions
- Instructions
- Operations
- Error messages

Prerequisites

Readers of this manual are assumed to be familiar with RA17K or *emIC-17K*.

Legend

This manual uses the following symbols and conventions:

...	:	Indicates that the preceding option can be repeated.
[ ]	:	The item enclosed in brackets is optional.
{|}	:	One of the characters or character strings, delimited by "|" in braces, must be selected.
" "	:	Indicates a character string.
' '	:	Indicates a character.
CR	:	Carriage return
L_F	:	Line feed
TAB	:	Horizontal tab
Δ	:	Indicates a space or tab.
≡	:	Indicates the contents corresponding to the expression.
< >	:	Character or characters to be specified as is, usually a title enclosed in <>.
xxxx	:	Indicates any character string.

Number representation systems :

Binary	:	xxxxB
Decimal	:	xxxx or xxxxD
Hexadecimal	:	xxxxH

File naming rule

[drive-name:][\directory-name\...]filename[.extension]

A file name may include a drive name and directory name(s). A path name includes a drive name and directory name(s) only.

Notation for error messages

In this manual, error messages are explained as follows:

(XXX:YYYY)

XXX: Message No.
YYYY: Explanation of the output message

XXX indicates the message number output when an error occurs and YYYY outlines the error message. Note that YYYY is not an error message itself; it just outlines an error message.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

[MEMO]

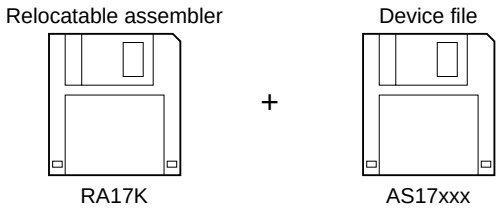
CHAPTER 1 OVERVIEW OF THE ASSEMBLER

1.1 FUNCTION OVERVIEW

The RA17K relocatable assembler is a common assembler designed to be used with 17K-series devices. RA17K processes source programs written in assembly language for the 17K series.

The RA17K.EXE file and a device file are necessary to execute RA17K. The device file contains information relating to a particular 17K-series device (target device), including debugging information and error information output at assembly time.

Figure 1-1. Files Required at Assembly Time



RA17K assembles an address at which an instruction or data is stored and creates object information. RA17K also supports a conventional absolute assemble mode in which the absolute address at which an instruction or data will be stored is determined at assembly time.

RA17K supports machine instructions and pseudo instructions of several types, which include macro pseudo instructions for creating a macro and conditional pseudo instructions. The operand to be coded in a machine instruction must be either a character string defined as a symbol or an expression that includes a symbol. Numerics should not be directly coded in the operand field (except as immediate data). Symbols are assigned attributes. The attributes of the operand are strictly examined.

Because RA17K provides built-in macro instructions that represent structured instructions or extended instructions that individually express two or more machine instructions for multiple nibble processing, the user can create a structured, easy-to-read program and debug the program efficiently.

1.1.1 Features

RA17K offers the following features:

(1) RA17K supports two assemble modes: relocatable mode and absolute mode.

(2) Because RA17K is a relocatable assembler, it can divide a source program into modules.

The use of two types of symbols is supported: A public symbol can be referenced outside a module; A local symbol can be referenced only within a module.

(3) Four types of symbols are added.

- Data type, data memory type, flag type, label type
- A symbol can consist of up to 253 characters.
- Shift JIS kanji code characters can be used (PC-9800 series only).
- A symbol type can be changed (type conversion function).

(4) RA17K provides a powerful macro function and a conditional assembly function.

A series of instruction blocks for fixed processing can be defined as a macro for repeated use. Part of a source program can be selectively assembled.

[Macro definition and conditional pseudo instructions]

- MACRO...ENDM
- REPT...ENDR
- IRP...ENDR
- IF...ENDIF
- IFCHAR...ENDIFC
- IFNCHAR...ENDIFNC
- IFSTR...ENDIFS
- CASE...ENDCASE

(5) RA17K has built-in macro instructions specifically for flag operation.

The 17K series does not itself support a bit manipulation instruction, this function instead being provided by a built-in macro instruction. This built-in macro instruction is incorporated into the main body of RA17K and need not be defined by the user.

* **[Built-in macro instructions]**

- SETn (Sets a flag.)
- CLRn (Resets a flag.)
- NOTn (Inverts a flag.)
- SKTn (Judges a flag. True)
- SKFn (Judges a flag. False)
- INITFLG (Initializes a flag.)
- SETX (Sets a flag. <Extended>)
- CLRX (Resets a flag. <Extended>)
- NOTX (Inverts a flag. <Extended>)
- SKTX (Judges a flag. True <Extended>)
- SKFX (Judges a flag. False <Extended>)
- INITFLGX (Initializes a flag. <Extended>)

(6) RA17K has built-in macro instructions for expanded machine instructions and structured instructions.

A single machine instruction of the 17K series can process an increased number of nibbles. An operation on two or more nibbles can be expressed in one instruction. Optimum instruction groups are automatically expanded.

* **[Extended machine instructions]**

- MOVX (Extended transfer instruction)
- LDX (Extended transfer instruction)
- STX (Extended transfer instruction)
- ADDX (Extended add instruction)
- ADDCX (Extended add instruction)
- ADDSX (Extended add instruction)
- ADDCSX (Extended add instruction)
- SUBX (Extended subtract instruction)
- SUBCX (Extended subtract instruction)
- SUBSX (Extended subtract instruction)
- SUBCSX (Extended subtract instruction)
- SKEX (Extended compare instruction)
- SKNEX (Extended compare instruction)
- SKGEX (Extended compare instruction)
- SKGTX (Extended compare instruction)
- SKLEX (Extended compare instruction)
- SKLTX (Extended compare instruction)
- RORCX (Extended rotate instruction)
- ROLCX (Extended rotate instruction)
- SHRX (Extended shift instruction)
- SHLX (Extended shift instruction)
- ANDX (Extended logical instruction)
- ORX (Extended logical instruction)

- XORX (Extended logical instruction)
- BRX (Extended branch instruction)
- CALLX (Extended function call instruction)
- SYSCALX (Extended system function call instruction)

* **[Structured instructions]**

- _IF..._ELSEIF..._ELSE..._ENDIF
- _WHILE..._ENDW
- _SWITCH..._CASE..._DEFAULT..._ENDS
- _REPEAT..._UNTIL
- _FOR..._NEXT
- _BREAK
- _CONTINUE
- _GOTO

(7) RA17K supports a wide range of options, all of which can be specified at activation.

(8) RA17K can be used with target devices with a ROM capacity of up to 64K words (eight segments installed).

(9) RA17K handles numeric data in 32-bit units.

1.2 ASSEMBLE MODES

RA17K supports two assemble modes: relocatable mode and absolute mode.

1.2.1 Relocatable Mode (Default)

Relocatable mode is the default assemble mode of RA17K.

In relocatable mode, absolute addresses are not determined at assembly time, relative addresses in units of sections being determined instead. The assembler creates an object module file but not an executable load module file (.ICE/.PRO). To create an executable load module file, a linker is started, then the object module file (.REL) created by the assembler is input.

The linker relocates the input object module file in units of sections. By means of this relocation, absolute addresses are determined and an executable load module file is created.

In relocatable mode, an error occurs if the following symbol or assemble-time variable is included in the program to be assembled.

- @AR_EPA0, @AR_EPA1 (EPA bit control symbol)
- ZZZEPA (assemble-time variable)

The following pseudo instructions produce different results in absolute mode and relocatable mode:

- ORG pseudo instruction
- CSEG pseudo instruction

1.2.2 Absolute Mode

RA17K enters absolute mode if an option is specified at activation. (The functions are almost the same as those of AS17K.)

When absolute mode is specified, absolute addresses are assigned at assembly time. Object module files are integrated into a single load module file by the link processing.

The linker does not relocate the input object module files, instead sequentially integrating the input object module files into an executable load module file.

In absolute mode, an error occurs if the following pseudo instruction is included in the program to be assembled.

* • ENSURE pseudo instruction (Can be included only in relocatable mode)

RA17K, in absolute mode, differs from AS17K in the following points:

- Evaluation value of a numeric or symbol
- ZZZn (assemble-time variable)
- ZZZSKIP (assemble-time variable)
- ZZZPRINT (assemble-time variable)
- END (pseudo instruction)
- EOF (control instruction)

Table 1-1. Differences between RA17K (in Absolute Mode) and AS17K

Function	AS17K	RA17K in absolute mode
Evaluation value of numeric or symbol**Note**	16 bits	32 bits
ZZZn	The value is passed to the next file.	The value is not passed to the next file.
ZZZSKIP	The value is passed to the next file.	The value is not passed to the next file.
ZZZPRINT	Bits 0 to 6 are used.	Information is added to bits 7 and 8 as well as to bits 0 to 6.
END	Cannot be omitted. (If omitted, an error message is output.)	Can be omitted. (If omitted, a warning message is output.)
EOF	If omitted, a warning message is output.	If omitted, a warning message is not output.

Note The evaluation value is expanded to 32 bits. The result of an operation is affected if it consists of 17 bits or more. When the length of an evaluation value exceeds 16 bits in absolute mode, a warning message is displayed.

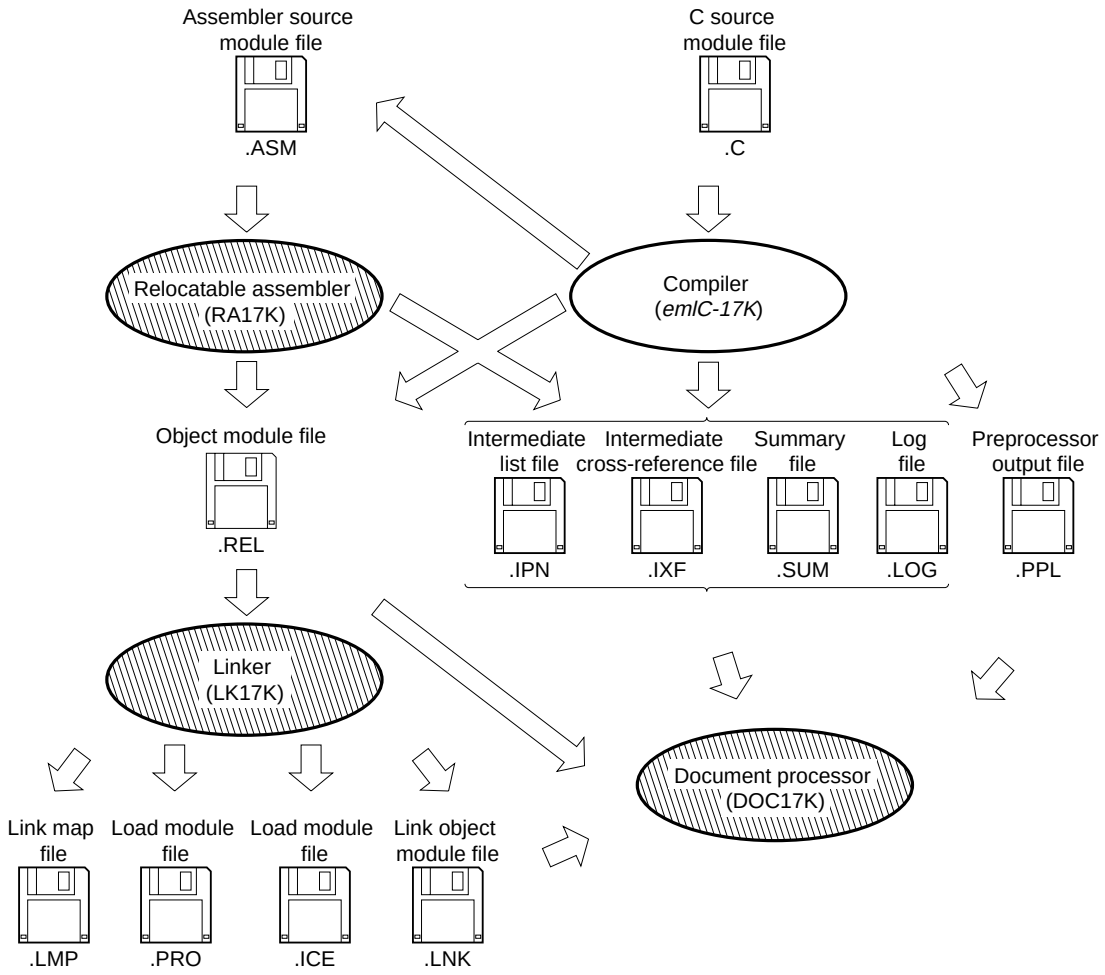
1.3 SYSTEM CONFIGURATION

RA17K is a two-pass relocatable assembler.

Codes are assembled in units of source modules.

Figure 1-2 shows an outline of the flow of software development with RA17K for the 17K series.

Figure 1-2. Flow of Software Development for the 17K Series



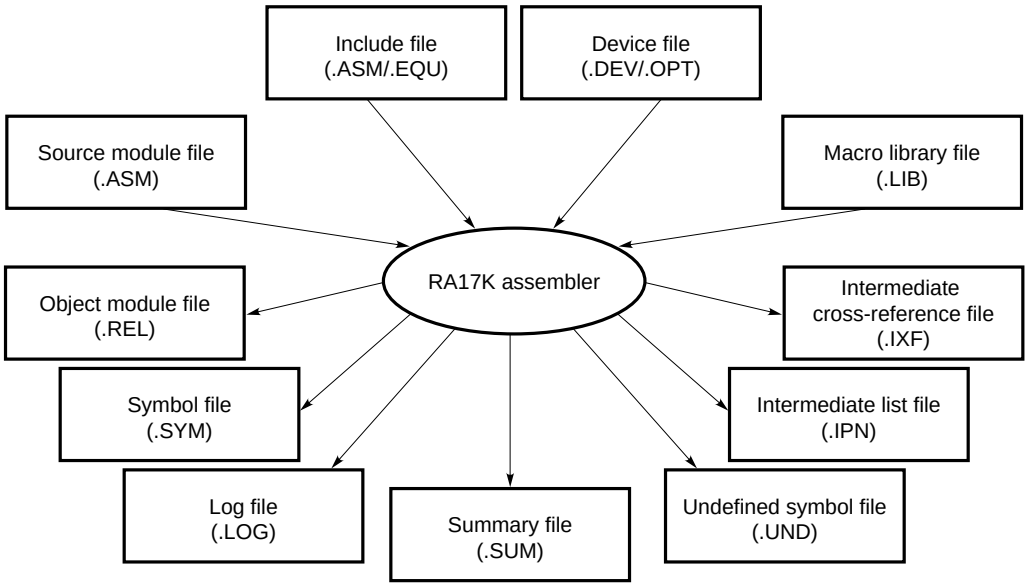
Remarks 1. For details of the files output by the document processor, refer to the **Document Processor (DOC17K) User's Manual**.

2. A shaded oval indicates a program included in the relocatable assembler package.

1.3.1 I/O Files

The figure below shows those files that are referenced and created by RA17K.

Figure 1-3. I/O File Configuration



[Input files]

- **Source module file (.ASM*)**
  A source program is written in this file. The user must create this file, using a text editor or other tools.

- **Include file (.ASM*/.EQU)**
  .ASM is another source program file. The file is included by the INCLUDE pseudo instruction. The user must create this file.
  .EQU is a file containing symbol definition statements only. The file is included by the INCLUDE pseudo instruction. The user must create this file. A symbol file (.SYM) is created from this file.

- **Device file (.DEV/.OPT)**
  In the .DEV file, information relating to items to be assembled (mnemonic, reserved word, debugging information, etc.) is stored. In the .OPT file, mask option definition information is stored. The file varies with the item to be assembled.

- **Macro library file (.LIB*)**
  Some devices have a macro library file. The user includes this file by using the INCLUDE pseudo instruction.

*: Default extension. The extension can be changed.

[Output files]

- **Object module file (.REL)**
  This file contains assembly information, object codes corresponding to a single source module file, and the *SIMPLEHOST* information.
  The file has the same name as the corresponding source module file. The file is not created if an error occurs at assembly time.

- **Symbol file (.SYM)**
  Based on the symbol definition file (.EQU), the symbol file is created to speed up assembly. The format of the symbol file is such that RA17K can easily process the data.
  Once a symbol file has been created, another symbol file can be created only after the symbol definition file has been updated.

- **Log file (source-module-file-name.LOG)**
  The log file contains messages to be displayed on the screen during execution. This file also outputs the assembly start time and end time.
  The file has the same name as the source module file. If assembly processing halts because of an error before all options have been analyzed, the file name becomes RA17K.LOG.

- **Summary information file (.SUM)**
  The file contains the summary information used by *SIMPLEHOST*. The SUMMARY pseudo instruction outputs the defined information.

- **Undefined symbol file (.UND*)**
  The output file contains undefined symbols corresponding to a single source module file.

- **Intermediate list file (.IPN*)**
  This list file contains the results of assembly. The document processor (DOC17K) must be used to obtain an absolute-address-based list.

- **Intermediate cross-reference file (.IXF*)**
  This file contains cross-reference information for symbols.
  This file is a binary file, not a list file (ASCII file). The document processor (DOC17K) must be used to obtain a complete cross-reference file.

In addition to the above files, a temporary work file is created on the disk during assembly. The work file is automatically deleted once assembly has been completed.

*: Default extension. The extension can be changed.

* 1.4 FILE CONFIGURATION

The file configuration described below is required to start RA17K.

(1) Main body of the assembler

- RA17K.EXE: Main body of RA17K (32-bit application)

(2) Attachment of files for DOS-Extender

Three files supported by Borland C are provided:

- 32RTM.EXE
- DPMI32VM.OVL
- WINDPMI.386

The above files can be distributed to end users.
The above files are also used with other 32-bit applications within the RA17K assembler package.

* 1.5 OPERATING ENVIRONMENT

RA17K operates under the following environment:

1.5.1 Hardware Environment

(1) Host machine

<1> PC-9800 series
<2> PC/AT™

(2) OS

MS-DOS™: Version 3.30 or later
PC DOS™: Version 5.02 or later
Windows 3.1, Windows 95

(3) Required memory

Conventional memory: 400K bytes or more
Protect memory: 2M bytes or more

(4) External storage (hard disk)

About 8M bytes of free space is required for installation.
At least 10M bytes is required for a work drive.

1.5.2 Software Environment

(1) Command line environment

As the memory driver, at least himem.sys or an equivalent is required.
Ensure normal operation under the following environments:

- himem.sys only
- himem.sys + emm386.exe

(a) PC-9800 series

Environment / OS	himem.sys only	himem.sys + emm386.exe
MS-DOS 3.30D	o **Note 1**	x **Note 1**
MS-DOS 5.00A	o	x
MS-DOS 6.2	o	o **Notes 2, 3**
Windows 3.1, Windows 95 (DOS prompt only) **Note 4**	o	o **Note 2**

Notes 1. MS-DOS 3.30D does not provide himem.sys and emm386.exe, so the use of the driver provided by Windows 3.1/95 is assumed.

2. At the end of emm386.exe, a /DPMI switch needs to be added.

3. Operation will be unpredictable if the DPMI server provided by DOS is installed.

4. To use RA17K under Windows, the project manager is required.

MS-DOS 5.00A and emm386.exe, provided by Windows 3.1/95, do not support the /DPMI function. So, to use UMB/EMS, combine it with a third-party product, or use the DOS window of Windows 3.1/95 instead of executing the command from the command line.

(b) PC/AT

Environment / OS	himem.sys only	himem.sys + emm386.exe
PC DOS 5.0	o	o
PC DOS 6.3	o	o
MS-DOS 6.2	o	o
MS-DOS 7.0	o	o
Windows 3.1, Windows 95 (DOS prompt only) **Note**	o	o

Note To use RA17K under Windows, the project manager is required.

(2) DOS window environment

(a) Common to PC-9800 series and PC/AT

Environment / OS	himem.sys only	himem.sys + emm386.exe
Windows 3.1	o **Note**	o **Note**
Windows 95	o	o

Note In the [386Enh] field of system.ini, windpmi.386, provided by the product, must be installed. For this installation, the installer provided by the product can be used.

1.6 ENVIRONMENTAL VARIABLES

RA17K supports the following environmental variables:

- DEV17K : Search path for device file
- INC17K : Search path for include file
- TMP : Search path for temporary file

If the path specified by the corresponding environmental variable cannot be found, the specification becomes invalid (no error occurs).

[Format]

```
DEV17K = [<path-name>][;<path-name>...]
INC17K = [<path-name>][;<path-name>...]
TMP = [<path-name>][;<path-name>...]
```

[Function]

The variable enables a file, necessary for assembly, in another directory to be referenced from the current directory.

If a path name is specified with environmental variable DEV17K or INC17K, a search is made for the device file or include file indicated by the path name.

If a path name is specified with environmental variable TMP, a temporary file is output to the file indicated by the path name.

Two or more path names can be specified by separating them with a semicolon (;). When multiple path names are specified, the first search is made based on the first path name. If the desired file cannot be found, the next search is made based on the second path name. If two or more path names are specified with TMP, only the first path name is valid.

- A search is made for a device file as described below. If the file cannot be found, an error occurs. See **Section 22.3.1** for details.

  (1) When a device file is specified without a path name
  - If environmental variable DEV17K is specified, that path is used as the basis for the search.
  - If environmental variable DEV17K is not specified, the current path is used as the basis for the search.

  (2) When a device file is specified with path name
  - The specified path is used as the basis for the search.

- A search is made for an include file as described below. If the file cannot be located, an error occurs. See **Section 19.1.1** for details.

  (1) When an include file is specified without a path name
  - If option -INCLUDE is specified, the path is used as the basis for the search.
  - If environmental variable INC17K is specified, the path is used as the basis for the search.
  - If neither of the above is specified, the path containing a source module file is used as the basis for the search.

(2) When an include file is specified with a path name

- The specified path is used as the basis for the search.

- A temporary file is generated under file name R$xxxxxx (xxxxxx is a random numeric).
  - If option -WORK is specified, the file is generated at the location indicated by the path.
  - If environmental variable TMP is specified, the file is generated at the location indicated by the path.
  - If neither of the above is specified, or if the specified path does not exist, the file is generated at the location indicated by the current path.

Remark See **Section 22.6** for details of the -INCLUDE and -WORK options.

* 1.7 LIMITATIONS

This section describes the limitations imposed by RA17K.

(1) A macro having body size (number of characters) of up to 64K bytes can be coded.

(2) The maximum number of symbols and macro bodies that can be registered varies with the memory capacity.

(3) Up to 64K bytes of information (characters) can be coded in a repetitive pseudo instruction IRP...ENDR or REPT...ENDR.

(4) Macro pseudo instructions, conditional assembly pseudo instructions (IF, for example), repetitive pseudo instructions (REPT, for example), and include files can be nested. They can be nested up to 40 levels deep. A built-in macro uses nesting that is one level deep.

(5) Include files can be nested up to eight levels deep. Any include files that are nested more than eight levels deep are not included.

* 1.8 INSTALLATION

For each host machine, RA17K is delivered using the distribution media and storage format indicated in Table 1-2.

Table 1-2. Distribution Media and Storage Format of RA17K

Host machine	OS	Distribution media	Storage format
PC-9800 series	MS-DOS (Version 3.30/5.00 or later) Windows (3.1/95) **Note**	3.5-inch, 2HD	MS-DOS
PC/AT	PC DOS (Version 5.00 or later) Windows (3.1/95) **Note**		PC DOS

Note To use RA17K under Windows, the project manager is required. RA17K operates under MS-DOS or PC DOS. So, when the project manager is not used, start RA17K under DOS.

Install the assembler package (RA17K) and project manager. When RA17K is used under Windows, the project manager is required.

The distribution media consist of four floppy disks.

Two installation methods are available:

Installation method	Installable program
Execute setup31.exe/setup95.exe under Windows.	Assembler package, project manager
Execute dosinst.bat under DOS.	Assembler package

[Execution example 1] Installation under Windows 95

The following is an execution example which reads the assembler package and project manager from drive A, and installs the assembler package and project manager into C:\nectools\bin.

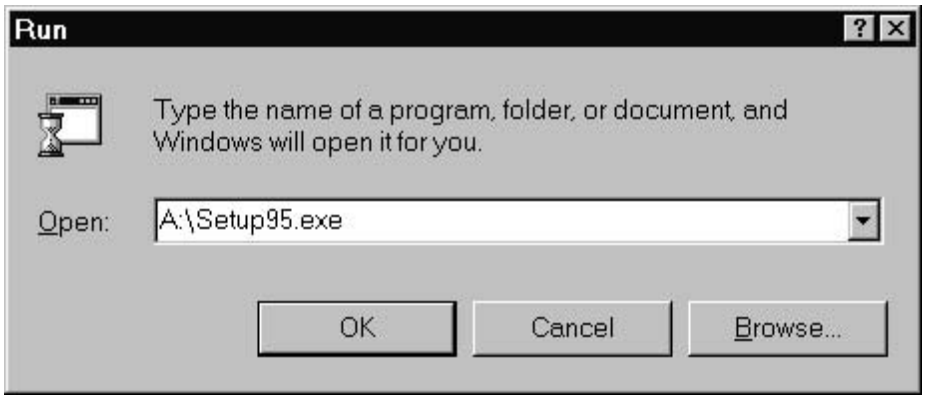
It is assumed that Windows 95 has already been started.

(1) Start the installer.

<1> Insert RA17K SETUP DISK#1 into the floppy disk drive.

<2> Select [Run] from the start menu.

<3> Enter the following in the “Open” field.



<4> Select “OK.” Then, the installer starts after setup initialization.

<5> Select “Continue.”

(2) Select an installation item.

<1> Select a product to be installed by clicking the corresponding check box.
By default, “Project Manager V2.13” and “17K Series Assembler Package V2.00” are selected for installation.

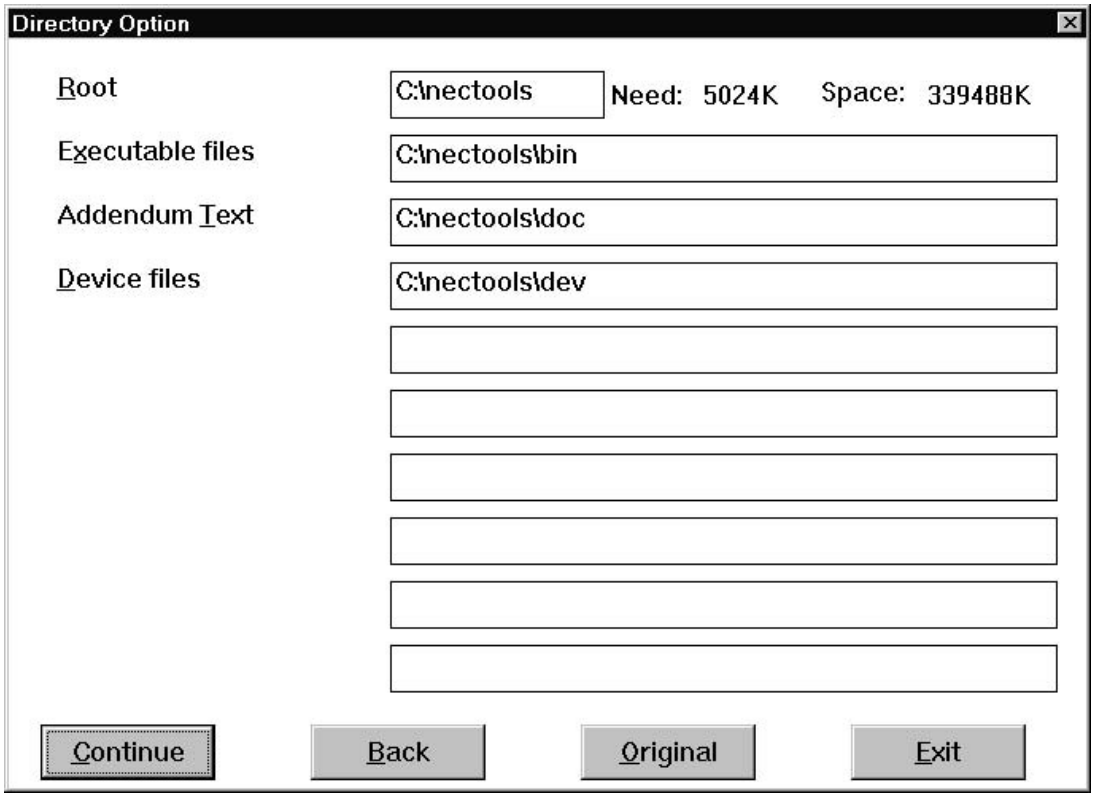
<2> After selecting a product to be installed, select “Continue.”

Remark Items that cannot be installed are grayed.

Caution When the assembler package is installed, the project manager must have already been installed, or it must be installed at the same time.

(3) Specify an installation directory.

<1> The Directory Option dialog box is displayed.



<2> Enter the desired installation directory in “Root.”

<3> Select “Continue.”

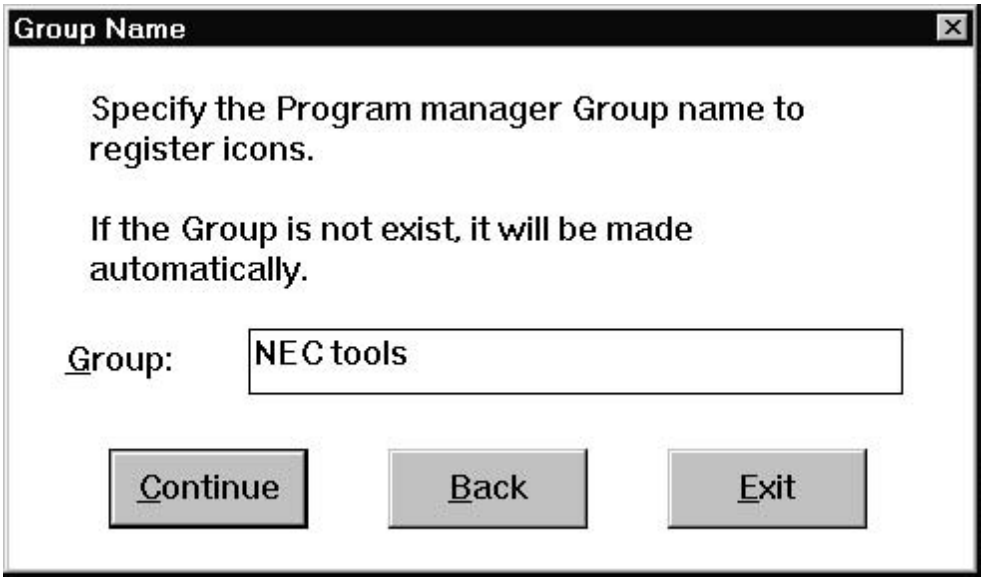
Remarks 1. When “Back” is selected, the display returns to the Products to Install dialog box.

2. When “Original” is selected, the default directory is selected. The default installation destination root is \nectools of the drive on which Windows 95 is installed. If a tool has already been installed with the installer, the root is selected. When a root change is made, the directories under the root are also changed accordingly.

3. When a supplement is unavailable, the directory in “Addendum Text” is grayed. When a supplement is available, the supplement is registered with an icon after installation. Users are recommended to read the supplement.

(4) Specify a registration group.

<1> The Group Name dialog box is displayed.



<2> Specify a desired registration group name in “Group.”
If the specified group does not yet exist, the group is newly created. If the specified group has already been registered using the installer, that group is used as is.

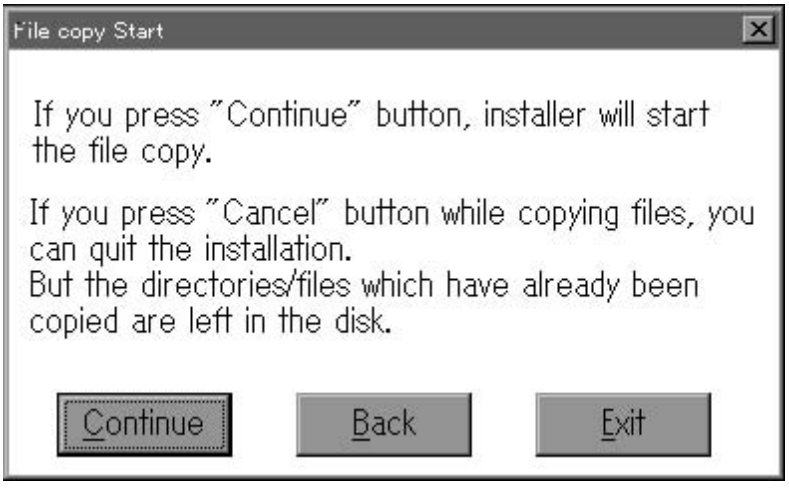
<3> Select “Continue.”

Remarks 1. When “Back” is selected, the display returns to the Directory Option dialog box.

2. When the project manager is not installed, the Group Name dialog box is not displayed.

(5) Start file copy operation.

<1> The File copy Start dialog box is displayed.



<2> When “Continue” is selected, file copy operation starts.

Remark When “Back” is selected, the display returns to the Group Name dialog box.

(6) Change the distribution media

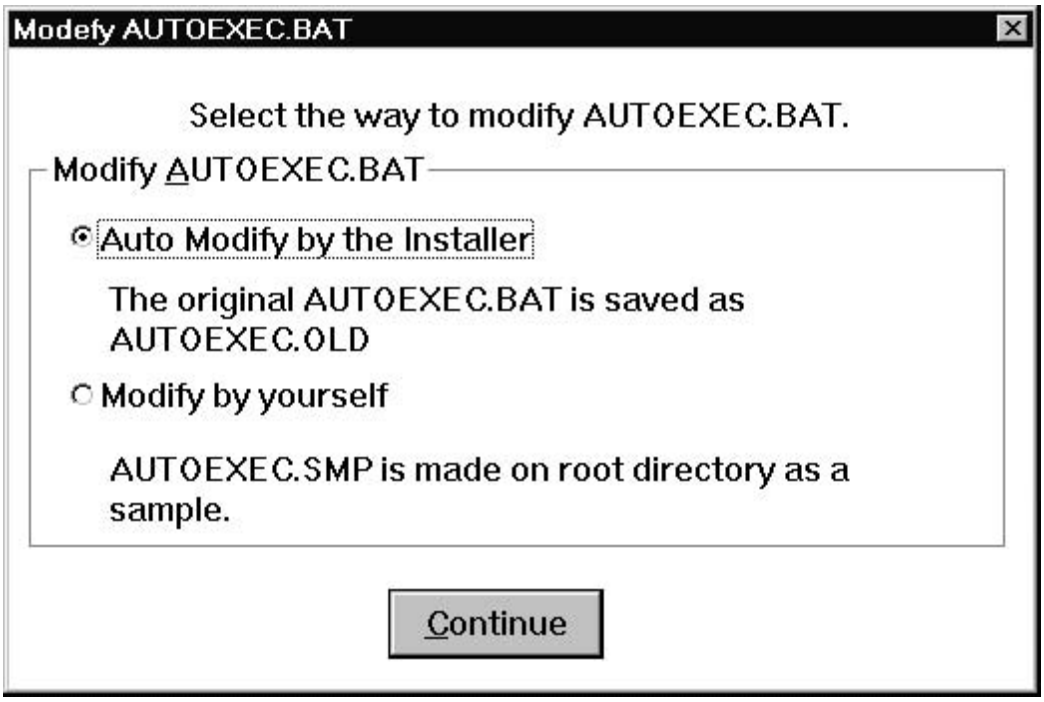
<1> When the following message is displayed, insert RA17K SETUP DISK#3 into the floppy disk drive.

<2> Similarly, insert RA17K SETUP DISK#4 into the floppy disk drive when prompted.

(7) The registered group and icon are created.

Caution The assembler package cannot be used under Windows, so that it is not registered as an icon.

(8) Modify AUTOEXEC.BAT.
Select “Auto Modify by the Installer” or “Modify by yourself” as the method for modifying AUTOEXEC.BAT.



- When “Auto Modify by the Installer” is selected
  AUTOEXEC.BAT on the drive where the Windows directory is placed is rewritten, then the original file is saved under the name AUTOEXEC.OLD.
- When “Modify by yourself” is selected
  As a rewrite sample, create AUTOEXEC.SMP under the root directory. When AUTOEXEC.SMP already exists, add the following:

  PATH c:\nectools\bin; %PATH%

(9) Terminate the installer.

<1> When “OK” is selected, the installer terminates.
<2> Restart the computer.

[Execution example 2] Installation under DOS
The following execution example reads the assembler package from drive A, then installs the executable format into C:\nectools\bin.
For installation, batch file dosinst.bat is executed.
The description format is as follows:

X>dosinst.batΔinstall-source-driveΔinstall-destination-driveΔinstall-destination-directory

Δ: One or more blanks

Example of execution

C >mkdir nectools;	Creates an installation destination directory.
C >a :;	Changes the current drive to the installation source drive.
A >dosinst.bat a : c : nectools;	Executes the batch file.

Once the execution of the batch file has been completed, rewrite autoexec.bat according to the contents of nectools\ra17k.add as follows:

PATH c:\nectools\BIN; %PATH%

Remark To stop the installation, press CTRL+C or ALT+C when the message “Press any key to continue...” appears.

CHAPTER 2 PROGRAM STRUCTURE

2.1 SPLIT ASSEMBLY OF A MODULE

A source program written in assembly language consists of one or more source module files. The assembler reads a source module file, which is a text file, assembles the file, then converts the file into an object module file that can be linked.

The source module file must be coded using the ASCII character set (or shift JIS character set, if Japanese is used). Other character types can be coded in the comment field and SUMMARY definition block. At the end of a statement, a line feed (L_F) code is necessary.

RA17K is a relocatable assembler. At assembly time, an object module is assigned a relative address. When the linker relocates the object module in units of sections, absolute addresses are assigned.

2.1.1 External Module Definition Reference Function

Pseudo instruction PUBLIC/EXTRN can reference a symbol defined by an external module.

At assembly time, the symbol defined by the external module is merely subjected to an attribute check and converted to an unresolved symbol. The object module file contains information relating to the unresolved symbol. The linker resolves the unresolved symbol, based on the information of the external module.

2.2 RELOCATION OF SECTIONS

In relocatable mode, the linker relocates a program module in units of blocks. A block is referred to as a section. A single section extends from one CSEG to another CSEG, from CSEG to END, or from CSEG to the end of the file.

An instruction to generate an object code can be coded only within the section block. (See **Chapter 10**.)

2.3 LAST INSTRUCTION OF A PROGRAM

The last instruction of a program is checked to prevent a program crash. If the ORG pseudo instruction causes a location address to be changed, the instruction preceding ORG is checked. If the instruction is other than a branch instruction, a program crash may occur. A warning is output to notify the user of this danger.

The check method depends on the assembly mode.

(1) Relocatable mode

Because a module is relocated in units of sections, a warning (W110: Invalid mnemonic in last of program) is output if an individual section does not end with a branch (BR), return (RET, RETI, RETSK), or data definition (DW, DB, DCP) instruction. The object, however, is output.

If the ORG pseudo instruction causes a location address in a section to be changed, the instruction preceding the ORG pseudo instruction is checked. The instruction is checked only when the location address is changed.

A warning is not output in the following case:

```
ORG    100H
ORG    200H    <- Because an instruction for generating an object code does not exist between ORG
                  100H and ORG 200H, no warning is output.
```

(2) Absolute mode

A warning (W110: Invalid mnemonic in last of program) is output if the instruction preceding the CSEG pseudo instruction or the last instruction of a user program is other than a branch (BR) instruction to a user area, return instruction (RET, RETI, RETSK), or data definition instruction (DW, DB, DCP). The object, however, is output.

If the ORG pseudo instruction causes a location address to be changed, the instruction preceding the ORG pseudo instruction is checked. The instruction is checked only when the location address is changed.

A warning is not output in the following case:

```
ORG    100H
ORG    200H    <- Because an instruction for generating an object code does not exist between ORG
                  100H and ORG 200H, no warning is output.
```

2.4 SEGMENT STRUCTURE

2.4.1 Segment Configuration

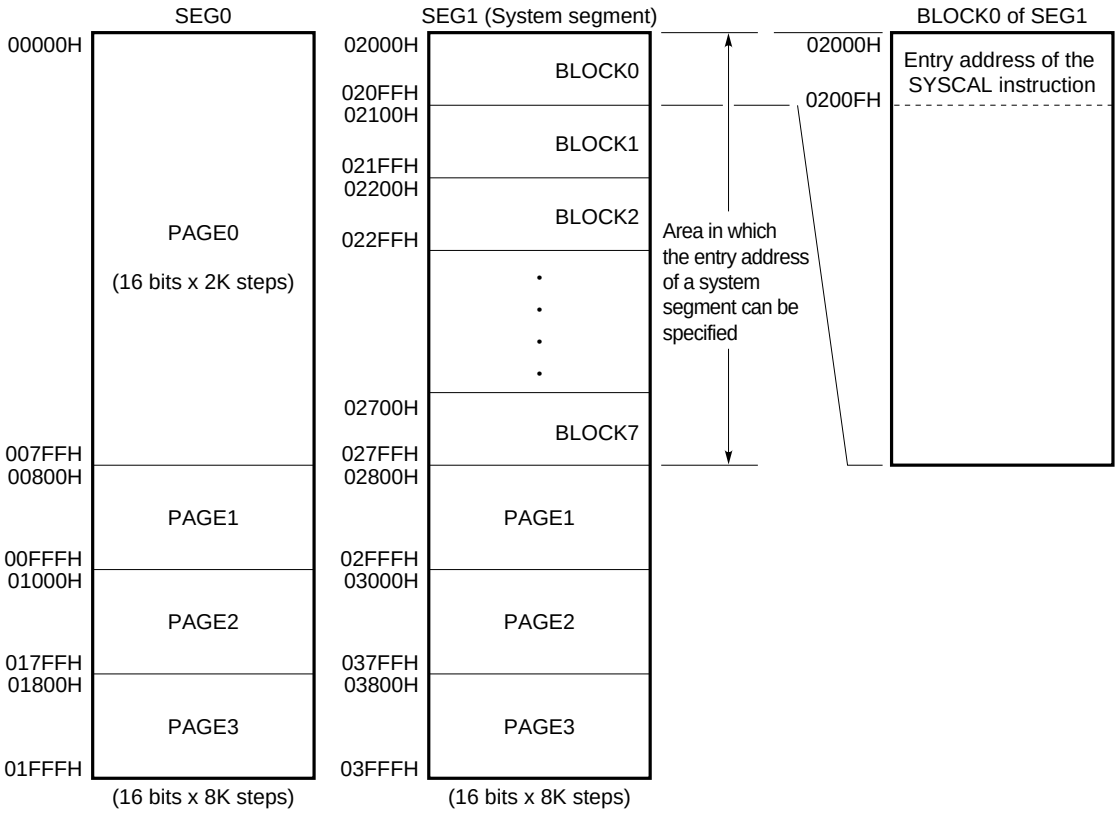
A segment consists of 2K-step pages. Some devices that have multiple segments can use the last segment as one that can be branched by the SYSCAL instruction. The last segment is called as the system segment. Page 0 of the system segment is divided into 256-step blocks (BLOCK0 to BLOCK7).

The 17K series supports two types of branch instructions: direct branch instructions and indirect branch instructions. Direct branch instructions (BR addr or CALL addr instructions) cannot invoke a branch to a point beyond the segment boundary. This is because the program counter capacity is 13 bits (8K steps). A segment is specified by the value of a segment register (SGR), separately from the program counter.

A branch to a point beyond the segment boundary is executed by an indirect branch instruction (BR @AR or CALL @AR instruction) that uses an address register (AR), which is one of system registers, or the SYSCAL instruction. A single SYSCAL instruction can invoke a branch to a system segment beyond a segment boundary. The first 16 words of each block of the system segment are used as the entry address for the SYSCAL instruction.

In the operand of the CALL addr instruction, the addresses of up to 11 bits can be specified. The branch destination of the CALL addr instruction must be within page 0.

Figure 2-1. Sample Segmented Address Space of the 17K Series (16K Steps Correspond to 32K Bytes)



2.5 EXTRA PROGRAM ADDRESS (EPA) STRUCTURE

When a program is debugged, the program size may temporarily exceed the ROM capacity of the target device. It would be inconvenient if this excess part could not be debugged. The IE-17K in-circuit emulator for the 17K series uses the program counter of the 17K-series device during debugging. When the ROM capacity of the target device is exceeded, the program counter will be too small to control the excess part of the program.

To overcome this problem, IE-17K incorporates an extra program address (EPA) bit. This enables the control of a program of up to double the ROM capacity of this chip.

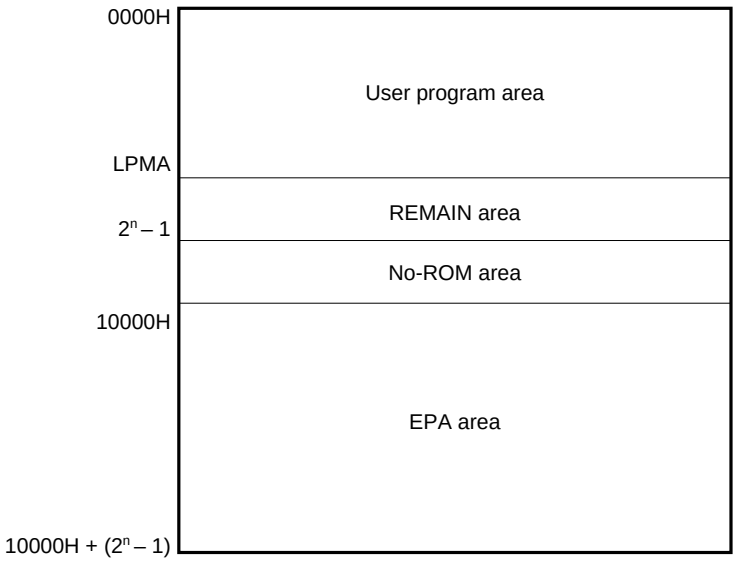
The function adds a single EPA bit to an address sent from the program counter of this chip and simultaneously adds one bit to the program counter. The address space that can be controlled is thus expanded to about double the ROM capacity of the chip. A program exceeding the ROM capacity can be debugged with IE-17K.

The EPA bit is added as a one-bit address, in the next-most significant bit position to the most significant bit of the ROM address of the target device. The ROM capacity which can be emulated is thus expanded to about double that of the target device. An address space in which the EPA bit is set to 1 is referred to as the EPA area and used as a patch area.

If the last ROM address of the target device (LPMA) is smaller than $2^n - 1$, the address space between LPMA to $2^n - 1$ becomes an emulation area that can be used without the EPA bit. This area is referred to as the REMAIN area and can be used as a patch area in the same way as the EPA area.

If a program exceeds the maximum valid address in ROM, LK17K causes an error. This error differs from other errors in that a normal object can be created in spite of the error. The program size should be reduced to a point where an error does not occur when the code of the ROM to be ordered is determined.

Figure 2-2. Address Space in a Segment of the 17K Series



2.5.1 Address Management

RA17K creates an object in an area outside normal program memory.

RA17K does not locate sections. RA17K does not output an error even if the size of an object exceeds the ROM capacity. Instead, the linker (LK17K) locates the sections.

Refer to the **LK17K User's Manual** for details of error output when a program exceeds the ROM capacity of a target device.

[MEMO]

CHAPTER 3 *SIMPLEHOST*™

3.1 OVERVIEW

SIMPLEHOST is a source level debugger designed to operate under Windows. *SIMPLEHOST* is available as optional support software for IE-17K and IE-17K-ET for the 17K series. *SIMPLEHOST* has functions for debugging object codes created by RA17K on the host machine, to send the codes to IE-17K or IE-17K-ET in real time, and to execute the codes on the SE board.

Refer to the user's manual provided with the optional *SIMPLEHOST* for details.

[MEMO]

CHAPTER 4 SOURCE PROGRAM CONFIGURATION

4.1 STATEMENT CONFIGURATION

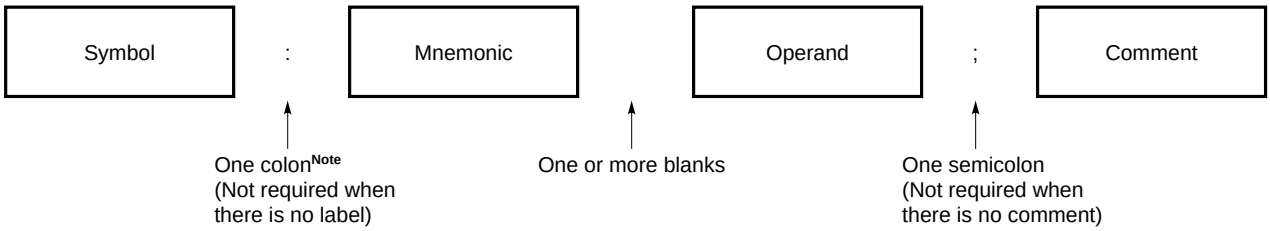
An RA17K source program consists of multiple statements.

A statement consists of four fields: symbol, mnemonic, operand, and comment, as shown below. It is terminated with an L_F (Line Feed). When using an editor to create a source program, each statement is usually terminated with a CR (Carriage Return)/L_F; however, the assembler ignores the CR.

Fields are separated with a blank (space or TAB), colon (":"), or semicolon (";"). These delimiters must be ASCII characters. Each line (from the beginning of the statement to the CR/L_F) can contain up to 255 characters. If the number of characters in a line exceeds 255, a warning (W157: Letters in a line are over 255) is issued; the first 255 characters are read and processed while the excess characters are ignored. The excess characters are not output to an intermediate list.

When writing a statement, a user can enter up to 254 characters per line, excluding the CR/L_F. Statements can be written in free form; the symbol, mnemonic, operand, and comment fields can begin at any column provided they appear in this order. Statements consisting only of a comment, as well as empty statements (lines having only a CR/L_F or lines filled entirely with blanks) can also be specified.

As a blank, specify either a space or TAB code.



Note Blank for a symbol definition pseudo instruction.

4.2 CHARACTERS

A source program is written using the characters listed below. Except for those listed in (4), the following are all ASCII characters.

(1) Alphabetic characters

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

Characters ?_ (underscore)

(2) Digits

0 1 2 3 4 5 6 7 8 9

(3) Special characters

Space		Field delimiter
@	(unit price symbol)	Indirect addressing specification symbol
,	(comma)	Operand delimiter
.	(period)	Bit delimiting operator
+	(plus)	Plus sign or addition operator
–	(minus)	Minus sign or subtraction operator
*	(asterisk)	Multiplication operator
/	(slash)	Division operator
(	(left parenthesis)	Used to change the order in which operations are performed.
)	(right parenthesis)	Same as above.
$	(dollar symbol)	Value of the location counter
=	(Equal sign)	Relational operator
;	(semicolon)	Comment start symbol
;;		Comment in a macro
:	(colon)	Label delimiter
'	(quotation mark)	Character constant start or end symbol
<		Relational operator
>		Relational operator
#		Immediate data specification symbol
&	(ampersand)	Specifies the concatenation of character strings in a macro.
%	(expression operator)	
TAB code		Equivalent to eight spaces.
L_F code		Statement termination symbol
CR code		Usually ignored by the assembler.

(4) Kanji, hiragana, and katakana (supported by only the PC-9800 series version)

Shift JIS code kanji, hiragana, katakana, and 8-bit JIS code katakana can be used.

No characters other than those listed in (1) through (4) can be used.

If a statement contains a character other than those listed above, an error (F082: Illegal character) occurs.

Note, however, that a comment field and a SUMMARY block defined with a SUMMARY pseudo instruction can contain any characters (visible characters). A comment field is terminated by a CR/L_F, whereas in a SUMMARY block, a CR/L_F can be entered in any position.

4.3 SYMBOL FIELD

In the symbol field, enter a symbol.

Symbols are divided into two categories: labels, to each of which a program memory address value is assigned to define the destination of a branch instruction; and names, each of which defines the data to be entered in the operand field. Labels and names are generically referred to as symbols.

If a character string entered at the beginning of a statement is terminated with a colon, that character string is assumed to be a label.

The value of the location counter of the program memory that stores the instruction immediately following a label is defined for that label. A label can also be defined with a symbol definition pseudo instruction (LAB), as described later.

When a label is entered, the label is registered in a symbol table, provided the description in the mnemonic field is correct, even if an error occurs in the operand field.

[Example]

```
ABCD:    IF    IF
```

This above example will produce a syntax error; however, because the descriptions in the symbol and mnemonic fields are correct, label ABCD will be registered.

A name is defined with a symbol definition pseudo instruction (such as DAT, FLG, or MEM). The evaluation value of the operand for the symbol definition pseudo instruction is assigned to the name. In addition to the evaluation value, variable length information can also be assigned by using the NIBBLEn or NIBBLEnV pseudo instruction, described later.

For a macro definition statement, enter a macro name in the symbol field. The block of macro definition pseudo instructions between MACRO and ENDM is defined for the macro name, as one procedure statement. A previously defined symbol name cannot be used as a macro name. If a macro name conflicts with a symbol name, an error occurs. (See **Chapter 16**.)

[Symbol coding rules]

(1) A symbol can consist of alphanumeric characters, underscore, ?, kanji, hiragana (Shift JIS code), and katakana (8-bit JIS or Shift JIS code). It must begin with an alphabetic character, underscore, ?, kanji, hiragana, or katakana character. A character string beginning with a digit is not recognized as a symbol. In a CASE to ENDCASE block, however, a numeric label beginning with a digit is supported.

(2) No limit is imposed on the number of characters constituting a symbol; however, a statement can contain no more than 255 characters, including the instruction, pseudo instruction, operand, and terminator (L_F). Note that one Shift JIS code character requires the same amount of space as two alphanumeric characters.

(3) The names used in symbol definition pseudo instructions and macro definition statements cannot be omitted. A name is terminated with a blank.

(4) A symbol cannot be defined more than once. If an attempt is made to redefine a symbol, an error (F057: Symbol multi defined) occurs. This rule does not apply to a symbol defined with a SET statement.
For an explanation of the scope of a local symbol defined in a macro or the scope of a symbol when the source program is divided into source modules, see **Section 16.4**.

(5) Reserved words cannot be used as symbols. Otherwise, an error (F037: Syntax error) occurs.

(6) Symbols are case-sensitive. For example, ABCD and abcd are recognized as being different symbols.

[Label coding rules]

(1) A label can consist of alphanumeric characters, underscore, ?, kanji, hiragana (Shift JIS code), and katakana (8-bit JIS or Shift JIS code). It must begin with an alphabetic character, underscore, ?, kanji, hiragana, or katakana character. A character string beginning with a digit is not recognized as being a symbol. In a CASE to ENDCASE block, however, the specification of a numeric label beginning with a digit is allowed.
If a label contains characters other than those described above, an error (F037: Syntax error) occurs.

(2) A label is terminated with a colon. One or more blanks (space or TABs) may be inserted between the label and the colon. (The blanks are ignored.)

(3) Labels can be written in a section or table block only. If a label is written elsewhere, an error (F146: Impossible to write out of section block) occurs.

(4) If an error occurs in a label description, and if an instruction for which object code will be created is written on the same line, the instruction will be nullified and the object code for an NOP instruction will be created; if a pseudo instruction is written on the same line, the pseudo instruction will be nullified.

(5) If a reserved word is written as a label, an error (F037: Syntax error) occurs.

4.3.1 Symbol Types

Symbols are assigned types. These types are listed below:

LAB type : LAB

DAT type : DAT

FLG type : FLG

MEM type: MEM
NIBBLE
NIBBLEn (1 ≤ n ≤ 8)
NIBBLEnV (2 ≤ n ≤ 8)

Caution For details of how to handle types, see the explanations of the symbol definition pseudo instruction and the type conversion instruction.

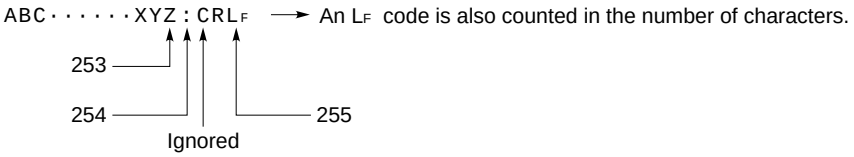
(1) Symbol examples

[Example 1] Label definition examples

Valid label	Invalid label	
F1F4:	1F4F:	... Begins with a digit.
LABEL:	LABEL	... A colon is missing.
HERE:	HE RE:	... There are blanks in the symbol.
ADDITION:	ADD:	... An instruction cannot be used.
TERMINATION:	END:	... A pseudo instruction cannot be used.

[Example 2] Limit imposed on the number of characters in a symbol

- A label can consist of up to 253 characters.



- For other symbols, see (2) in the Symbol coding rules.

[Example 3] Symbol definition example

A previously defined symbol can be entered as an operand in a symbol definition statement.
In this case, the same attributes and value as those specified for the symbol entered in the operand field are defined for the symbol in the symbol field.

```
 Symbol    Mnemonic    Operand    Comment

 ABC          DAT        0300H
 XYZ          DAT        ABC        ; 0300H is defined for XYZ.
```

[Example 4] Symbol multi-definition

If the same name is defined more than once, the second and any subsequent definitions will cause an error (F057: Symbol multi defined). Only the first definition is valid.

```
 Symbol    Mnemonic    Operand    Comment

 TURN:        NOP
               :
               :
               :
 TURN         MEM        0.01H      ; Error (F057:  Symbol multi defined)
              ADD        TURN, #1   ; Error (F011:  Illegal first operand type)
```

[Example 5] Symbols defined with SET pseudo instructions

The value of the symbol defined with a SET pseudo instruction can be changed. A SET pseudo instruction is used to change as well as set the value.

```
 Symbol    Mnemonic    Operand    Comment

 Counter      SET           3       ; (Counter=3)
              IF         Counter=3  ; The IF block is expanded.
               :
               :
               :
              ENDIF
 Counter      SET           5       ; (Counter=5)
              IF         Counter=3  ; The IF block is not expanded.
               :
               :
               :
              ENDIF
```

Because a symbol defined with a SET pseudo instruction can be redefined, the symbol can be used as a variable that is meaningful at assembly time only. This user's manual refers to such a variable as an assemble-time variable.
The contents of an assemble-time variable are referenced with an instruction such as the IF pseudo instruction, and used for conditional assembly, etc.

4.3.2 Reserved Words

Reserved words are those words which are defined by the system. They include instruction, pseudo instruction, register, and operator names. Such names can be used only exactly as they are defined; they cannot be redefined as symbols. As will be described later, however, the values of assemble-time variables (reserved words having prefix ZZZ) can be changed with a SET pseudo instruction. Also, one character string can be replaced with another by using a LITERAL pseudo instruction.

- Reserved words are not case-sensitive. For example, a reserved word is handled in the same way regardless of whether it is entered in uppercase or lowercase. (Ordinary symbols are case-sensitive. For example, sym and SYM are recognized as being different symbols.)
- Reserved words vary with the hardware specifications and, therefore, differ with the product type. A list of reserved words is provided in the data sheet and user's manual for each device.

Reserved words

_BREAK	_CASE	_CONTINUE	_DEFAULT
_ELSE	_ELSEIF	_ENDIF	_ENDS
_ENDW	_FOR	_GOTO	_IF
_NEXT	_REPEAT	_SWITCH	_UNTIL
_WHILE	ADDCSX	ADDCX	ADDSX
ADDX	ANDX	AR_EPA0	AR_EPA1
BANK0	BANK1	BANK2	BANK3
BANK4	BANK5	BANK6	BANK7
BANK8	BANK9	BANK10	BANK11
BANK12	BANK13	BANK14	BANK15
BELOW	BRX	C14344	C4444
CALLX	CASE	CLR1	CLR2
CLR3	CLR4	CLRX	CSEG
DAT	DB	DCP	DW
EJECT	ELSE	END	ENDCASE
ENDIF	ENDIFC	ENDIFNC	ENDIFS
ENDM	ENDOP	ENDP	ENDR
ENDSUM	ENSURE	ENTRY	EOF
EQ	EXIT	EXITR	EXTRN
FLG	GE	GLOBAL	GT
IF	IFCHAR	IFNCHAR	IFSTR
INCLUDE	INITFLG	INITFLGX	INV
IRP	LAB	LBMAC	LE
LFCOND	LIST	LITERAL	LMAC
LT	MACRO	MEM	MOD
MOVTX	MOVX	NE	NIBBLE

NIBBLE1	NIBBLE2	NIBBLE3	NIBBLE4
NIBBLE5	NIBBLE6	NIBBLE7	NIBBLE8
NIBBLE2V	NIBBLE3V	NIBBLE4V	NIBBLE5V
NIBBLE6V	NIBBLE7V	NIBBLE8V	NOBMAC
NOCHANGE	NOLIST	NOMAC	NOT
NOT1	NOT2	NOT3	NOT4
NOTX	OBMAC	OMAC	OPTION
ORG	ORX	OTHER	PAGE0
PAGE1	PAGE2	PAGE3	PUBLIC
PURGE	REPT	ROLCX	RORCX
SBMAC	SET	SET1	SET2
SET3	SET4	SETAR	SETBANK
SETIX	SETMP	SETRP	SETX
SFCOND	SHL	SHLX	SHR
SHRX	SKEX	SKF1	SKF2
SKF3	SKF4	SKFX	SKGEX
SKGTX	SKLEX	SKLTX	SKNEX
SKT1	SKT2	SKT3	SKT4
SKTX	SMAC	SUBCSX	SUBCX
SUBSX	SUBX	SUMMARY	SYS
SYSCALX	TABLE	TITLE	UNKNOWN
UNLITERAL	VBMAC	VMAC	XORX
ZZZERROR	ZZZMCHK	ZZZMSG	ZZZOPT

4.4 MNEMONIC FIELD

In the mnemonic field, enter an instruction, pseudo instruction, or macro reference statement.

For an instruction that requires an operand, one or more blanks (spaces or TABs) are required to separate the mnemonic field from the operand field.

Assembly processing creates the object code for an instruction written in a mnemonic field. A pseudo instruction is an instruction for the assembly processing.

4.5 OPERAND FIELD

In the operand field, enter the operand for the instruction, pseudo instruction, or macro reference (including a built-in macro instruction).

Some 17K series instructions require no operand. Others, however, require that an operand be specified. Some pseudo instructions allow more than one operand to be entered. To enter two or more operands, delimit them with a comma (,)**Note**. No limit is imposed on the number of operands; however, the statement must fit on one line (255 characters).

For those instructions that require the specification of operand(s), the number of operands and evaluation values are examined.

One or more blanks (spaces or TABs) must be inserted between a mnemonic field and an operand field.

As the operand for a pseudo instruction, an external definition symbol defined in another module cannot be entered.

Note For the CSEG pseudo instruction only, a blank is used as the operand delimiter.

4.5.1 Operand Field Coding Format

(1) Constants

Constants are divided into numeric constants which consist entirely of digits, and character constants which consist entirely of characters.

Numeric constants include binary, octal, decimal, and hexadecimal constants. They are entered using single-byte characters.

If a numeric constant consists of more than 32 bits, an error (F164: The constant is over 32 bits) occurs. If any error other than this occurs, and if the constant is entered as the operand of an instruction for which object code will be created, the object code for an NOP instruction will be created; if it is entered as the operand of a pseudo instruction, the pseudo instruction will be nullified.

(a) Binary constants

A binary constant is identified by being suffixed with a single-byte B.

The B may be either uppercase or lowercase. If a binary constant contains digits other than 0 and 1, an error (F044: Invalid value) occurs.

[Example]

```
1011B
1011b       ; The B may also be in lowercase.
```

(b) Octal constants

An octal constant is identified by being suffixed with a single-byte O or Q.
The O or Q may be either uppercase or lowercase.
If an octal constant contains digits other than 0 through 7, an error (F044: Invalid value) occurs.

[Example]

```
12345670      ···A single-byte O is added to the end.
1234Q         ···A single-byte Q is added to the end.
```

(c) Decimal constants

A decimal constant is identified by being suffixed with a single-byte D or nothing.
The D may be either uppercase or lowercase.
If a decimal constant contains digits other than 0 through 9, an error (F044: Invalid value) occurs.

[Example]

```
1234567890    ···Nothing is added to the end.
1234567890D   ···A single-byte D is added to the end.
```

(d) Hexadecimal constants

A hexadecimal constant is identified by being suffixed with a single-byte H. It must begin with a character between 0 and 9. If it must begin with a character between A and F, it must be prefixed with a 0. (A constant that does not begin with a character between 0 and 9 is recognized as a symbol.)
The H may be specified in either uppercase or lowercase.
If a hexadecimal constant contains digits other than 0 through 9 and A through F, an error (F044: Invalid value) occurs.

[Example]

```
147H
1ABCDEFH
ABCDH    ; Not recognized as a hexadecimal constant because it begins
           with a character other than 0 through 9.  0 must be added
           to the beginning to change it to 0ABCDH.
```

(e) Character constants

A character constant consists of characters enclosed in single quotation marks (').

A <character-string> enclosed in single quotation marks is called a character constant. The <character-string> can contain any characters other than a CR/L_F. In the same way as numeric constants, character constants have evaluation values (ASCII code values).

[Format]

```
'<character-string>'
```

[Example]

```
' A'       : 41H
' ' ' '    : 27H (Two single quotation marks are recognized as a single quotation mark.)
' A' ' '   : 4127H
'   '      : 20H (space code)
'  <'      : 203CH
' ABCD'    : 41424344H
```

[Notes]

(1) When a character constant is written in a macro, if the <character-string> contains the same character string as a formal parameter, the assembler does not recognize it as the formal parameter.

```
LOC.  OBJ.  M  I   STATEMENT
                            MACW  MACRO   X
                                  ZZZMSG  ' Invalid value X'
                            ENDM
                                  MACW    %MEM1
                1                 ZZZMSG  ' Invalid value X'
                                                          ↑
                                                This parameter is not replaced.
```

(2) When a character constant is entered as the actual parameter of a macro, only the character string enclosed in single quotation marks is passed. That is, the single quotation marks at both ends are removed.

```
LOC.  OBJ.  M  I   STATEMENT
                            MACC  MACRO   X
                                  ZZZMSG  X
                            ENDM
                                  MACC    ' Invalid value'
                1                 ZZZMSG  Invalid value    <- Error
                                  MACC    ''' Invalid value'''
                1                 ZZZMSG  ' Invalid value' <- Normal processing
```

(2) $ (location counter)

$ returns the value of the location counter. That is, it indicates the program memory address of the instruction for which $ is written.

[Example]

```
100          MOV    R0, #20H
101 LOOP:    ADD    R2, #30H
102          BR     $–1
```

In the above example, $ indicates address 102H and, therefore, BR $–1 is a jump instruction that causes a jump to address 101H. BR $–1 is equivalent to BR LOOP, in which label LOOP is used.

(3) Symbols

When a symbol is entered in an operand field, the value assigned to the symbol (label or name) is assumed as the value of the operand.

[Example]

```
Here:    BR    There
          :
          :
          :
There:   RET
```

(4) Expressions

A combination of constants, $, and symbols linked by operators is called an expression. There are 17 operators. Each is assigned a priority.

Bit delimiting operators, necessary to represent memory and flag addresses, can be written as part of an expression. (See **Section 4.7**.)

4.6 COMMENT FIELD

A comment field begins with a semicolon (;) and ends with a line feed code (L_F).

Comments can be provided to improve the readability of program processing, etc. Comments are ignored during assembly processing and are output to the assembly listing exactly as entered.

The document creation function and memory map creation function supported by the document processor can extract comments entered in the lines of symbol definition pseudo instructions and those of specific instructions (such as branch instructions) and expand them into a list as symbol information. (The document creation function and memory map creation function automatically extract symbols and comments from a source program to create a list.)

A comment can start in any column, provided it appears after the symbol, mnemonic, and operand fields. The characters enclosed between a semicolon and a L_F are recognized as a comment and are not subjected to assembly processing. Note that it is also possible for a line to begin with a semicolon (i.e., the whole line becomes a comment).

Remark A comment can also be entered using the SUMMARY pseudo instruction of the document creation function. If a source program is written hierarchically, writing comments using SUMMARY pseudo instructions can assist in maintaining the program.

4.7 EXPRESSIONS AND OPERATORS

4.7.1 Expressions

A character or numeric expression containing symbols, constants, and operators in its operand field is referred to as an expression.

Expressions are classified as data types (DAT), data memory types (MEM), flag types (FLG), and label types (LAB).

The combinations of types that can be processed are shown below, together with relevant notes.

- Type and numeric value : The result of an operation will have the same attributes as the type.
- Type and type : Symbols of different types cannot be processed.
- Numeric value and numeric value : The result of an operation will be of data type.**Note**

Note A numeric value is processed as a data type.
To process symbols of different types, ensure that they are of the same type, using the type conversion function if necessary.

The following expressions can be written.

For information on the types of symbols used in each expression, see **Chapter 8**.

```
<expression (DAT-type) >  ≡ <numeric-value>
                          ≡ <DAT-type-symbol>
                          ≡ <numeric-value> <operator> <numeric-value>
                          ≡ <DAT-type-symbol> <operator> <numeric-value>
                          ≡ <expression (DAT-type) > <operator> <numeric-value>
                          ≡ <expression (DAT-type) > <operator> <DAT-type-symbol>

<expression (MEM-type) >  ≡ <expression (DAT-type) >. <expression (DAT-type) >Note
                          ≡ <MEM-type-symbol>
                          ≡ <MEM-type-symbol> <operator> <expression (DAT-type) >

<expression (FLG-type) >  ≡ <expression (DAT-type) >. <expression (DAT-type) >.
                            <expression (DAT-type) >Note
                          ≡ <expression (MEM-type) >. <expression (DAT-type) >Note
                          ≡ <FLG-type-symbol>
                          ≡ <FLG-type-symbol> <operator> <expression (DAT-type) >

<expression (LAB-type) >  ≡ <numeric-value>
                          ≡ <LAB-type-symbol>
                          ≡ <numeric-value> <operator> <numeric-value>
                          ≡ <LAB-type-symbol> <operator> <numeric-value>
                          ≡ <expression (LAB-type) > <operator> <numeric-value>
                          ≡ <expression (LAB-type) > <operator> <LAB-type-symbol>
```

A <numeric-value> can be coded in binary, octal, decimal, or hexadecimal.

Note An <expression> with a bit delimiting operator (.) can be entered as the operand for the MEM pseudo instruction and FLG pseudo instruction only. If an <expression> with a bit delimiting operator is entered as the operand for a pseudo instruction other than MEM and FLG or for a mnemonic, an error (F037: Syntax error) occurs.

[Notes]

(1) If an <expression> contains symbols of different types, an error (F045: Invalid type) occurs. If, however, the <expression> is entered as the operand of an instruction for which object code will be created, an error (F011: Illegal first operand type) occurs if it is the first operand; similarly, an error (F012: Illegal second operand type) occurs if it is the second operand.
Operations on symbols of data and data memory types; data and label types; and data and flag types do not cause errors.

(2) When an error occurs in an <expression> description, if the description is entered as the operand of an instruction for which object code will be created, the instruction will be nullified and the object code for an NOP instruction will be created; if it is entered as the operand of a pseudo instruction, the pseudo instruction will be nullified.

(3) In absolute mode, if the final evaluation value or an intermediate result of an <expression> exceeds 16 bits, a warning (W111: The result is over 16 bits) is issued.

(1) Data type (DAT type) expression

A data type expression is used to represent 32-bit data. If the result of an expression exceeds 32 bits, the 33rd and subsequent bits are ignored.

Immediate data, i.e., a data type expression preceded by a #, entered as the operand of an instruction, represents 4-bit data. If the result of this expression exceeds 4 bits, an error (F015: Illegal second operand value) occurs. (For an extended instruction, the immediate data can be 32-bits data.)

A data type expression can use constants and data type symbols.

To use a symbol of a type other than data type in a data type expression, convert the type.

[Example]

```
S1      CSEG
Count   DAT     0256H               ; <1>
MEM1    MEM     0.00H               ; <2>
        :
        MOV     MEM1, #Count/82H    ; <3>
        :
        ADD     MEM1, #Count*4H     ; <4>
                ;          ↑
                ; Causes an error
        END
```

[Description]

<1> The value of 0256H is assigned to the name, Count.

<2> Bank 0 and data memory address 00H are assigned to the name, MEM1.

<3> Count/82H (256H/82H=4H) is stored in MEM1. Count/82H is a data type expression.

<4> Count*4H is equivalent to 256Hx4H in this example, the result of which exceeds 4 bits. Therefore, an error (F015: Illegal second operand value) occurs.

(2) Data memory type (MEM type) expression

A data memory type expression is used to represent a data memory address.

A data memory type expression can use a position delimiting symbol ".". Once an operation has been executed, only the low-order 12 bits of the data are valid.

The symbol types that may be specified in a data memory type expression are the data memory and data types.

[Example]

```
S2      CSEG
MEM4    MEM     0.10H                    ┐
MEM5    MEM     0.20H                    │ <1>
CONST1  DAT     2H                       │
CONST2  DAT     4H                       ┘
        ⋮
        MOV     MEM4+4H,#CONST1          ; <2>
MEMA    MEM     CONST1+3H.CONST2+2H      ; <3>
        ⋮
        END
```

[Description]

<1> MEM4, MEM5, CONST1, and CONST2 are defined with symbol definition pseudo instructions.

<2> The expression MEM4+4H indicates bank 0 and data memory address 14H. MEM4 is a data memory type symbol.

<3> The expressions CONST1+3H and CONST2+2H indicate bank 5 and data memory address 06H. Thus, MEMA is defined as bank 5 and data memory address 06H.

(3) Flag type (FLG type) expression

A flag type expression is used to represent a flag.

A flag type expression does not permit an operation on flag type symbols. Only the operation in the range specified with a position delimiting symbol (.) is valid. The allowable symbol types are the data and data memory types.

[Example]

```
S3      CSEG
MEM6    MEM     0.13H                              ┐
CONST3  DAT     0H                                 │ <1>
CONST4  DAT     14H                                │
CONST5  DAT     3H                                 ┘
        :
FLAG1   FLG     MEM6.0H                          ; <2>
FLAG2   FLG     CONST3+2H.CONST4+6H.CONST5       ; <3>
        :
        END
```

[Description]

<1> MEM6, CONST3, CONST4, and CONST5 are defined with symbol definition pseudo instructions.

<2> Bank 0, data memory address 13H (MEM6), and bit position 0 (LSB) are assigned to the name, FLAG1. In this example, MEM6 is a data memory type symbol.

<3> Bank 2, data memory address 1AH, and bit position 3 (MSB) are assigned to the name, FLAG2. In this example, CONST3, CONST4, and CONST5 are data type symbols.

(4) Label type (LAB type) expression

A label type expression is used to represent a program memory address (the value of the location counter). A label type expression can use constants and label type symbols.
To use a symbol of a type other than label type in a label type expression, convert the type.
When a label type symbol is defined with a LAB pseudo instruction, the following occur if the evaluation value of the expression falls outside the ROM range:

- **If the evaluation value is in the EPA area**
  A warning (W153: The address is in EPA area) is issued.

- **If the evaluation value falls outside the ROM and EPA areas**
  An error (F152: The address is out of ROM) occurs.

[Example]

```
S4              CSEG
Data table 1    LAB       0300H                 ; <1>
                :
                ORG       Data table 1          ; <2>
Table area 1:   DB        00H,48H
                :
                ORG       Data table 1 + 20H    ; <3>
Table area 2:   DB        10H,52H
                :
                ORG       Data table 1 + 40H    ; <4>
Table area 3:   DB        50H,60H
                :
                END
```

[Description]

<1> The value of 0300H is assigned to the label type symbol, Data table 1.

<2>, **<3>**, **<4>** The top address of each table area is defined with a label type expression.

4.7.2 Operators

(1) Outline

The operators of the RA17K assembly language are divided into five types. Priorities are set for each of these operators.

- **Arithmetic operators**
  +, –, *, /, MOD (remaindering)

- **Logic operators**
  OR, AND, XOR, NOT

- **Relational operators**
  EQ, NE, LT, LE, GT, GE
  =, <>, <, <=, >, >=

- **Shift operators**
  SHR, SHL

- **Unary operators**
  +, –

(2) Operator priorities

Priorities are set as shown in the table below. () can change the order in which operations are performed. If operators having the same priority exist in an expression, they are performed from left to right.

Table 4-1. Priorities of Operators

Priority	Operator(s)
1	() (operator order specification symbols)
2	NOT, + (unary operator), – (unary operator), type conversion function, .TYPE., .DEF., .EV.
3	*, /, MOD, SHL, SHR
4	+, – (arithmetic operator)
5	AND
6	OR, XOR
7	EQ, NE, LT, LE, GT, GE, =, <>, <, <=, >, >=

4.7.3 Arithmetic Operators

(1) Addition operator (+)

[Format]

<expression-1>+<expression-2>

[Function]

This operator adds the values of <expression-1> and <expression-2> together.

[Explanation]

If the result of the operation exceeds the 32-bit range (-2^{31} to 2^{31}) including the sign bit, the high-order bits beyond the 32-bit limit are truncated.

[Example]

```
START   DAT     4H                              ┐
OFFSET  DAT     3H                              │
STEP    DAT     2H                              │ <1>
                                                │
R1      MEM     0.01H                           ┘
        ⋮
        MOV     R1,#START + OFFSET              ; <2>
LOOP1:  ⋮
        ADD     R1,#STEP                        ; <3>
        SKF1    CY                              ; <4>
        BR      LOOP1END
        ⋮
        BR      LOOP1
LOOP1END:
```

[Description]

<1> Symbols are defined.

<2> As an initial value, START+OFFSET (07H) is stored in R1.

<3> STEP is added to R1.

<4> If there is a carry, a jump to LOOP1END occurs.

(2) Subtraction operator (–)

[Format]

<expression-1>–<expression-2>

[Function]

This operator subtracts the value of <expression-2> from that of <expression-1>.

[Explanation]

If the result of the operation exceeds the 32-bit range (-2^{31} to 2^{31}) including the sign bit, the high-order bits beyond the 32-bit limit are truncated.

[Example]

```
TABLE end       LAB     100H                    ┐
                                                ├ <1>
TABLE area      LAB     40H                     ┘
                ⋮
                ORG     TABLE end–TABLE area  ; <2>
TABLE start:
                DW      0445H                   ┐
                                                ├ <3>
                DW      5637H                   ┘
                ⋮
                ORG     TABLE end
                ⋮
```

[Description]

<1> Symbols are defined.

<2> The start address of the table area is set in the TABLE end–TABLE area (0C0H).

<3> Define data.

(3) Multiplication operator (*)

[Format]

<expression-1>*<expression-2>

[Function]

This operator multiplies the value of <expression-1> by that of <expression-2>.

[Explanation]

If the result of the operation exceeds the 32-bit range (-2^{31} to 2^{31}) including the sign bit, the high-order bits beyond the 32-bit limit are truncated.

[Example]

```
Table      LAB       100H                    ┐ <1>
Block      LAB       10H                     ┘

           :

           ORG       Table
Table area 1:                                 ┐
           :                                  │
           ORG       Table+Block              │
Table area 2:                                 │ <2>
           :                                  │
           ORG       Table+(Block*2)          │
Table area 3:                                 │
           :                                  │
           ORG       Table+(Block*3)          ┘
Table area 4:
           :
```

[Description]

<1> Symbols are defined.

<2> The uppermost address of each table area is defined in program memory.

(4) Division operator (/)

[Format]

<expression-1>/<expression-2>

[Function]

This operator divides the value of <expression-1> by that of <expression-2>.

[Explanation]

If the result of an operation exceeds the 32-bit range (-2^{31} to 2^{31}) including the sign bit, the high-order bits beyond the 32-bit limit are truncated.
If <expression-2> is equal to 0, an error (F044: Invalid value) occurs and the result of the operation is set to 0.
If the result is not an integer, the decimal part is truncated.

[Example]

```
Table area      LAB     40H                                  ┐
                                                             │ <1>
Table start     LAB     200H                                 ┘
                ⋮
                ORG     Table start+(Table area/4H)          ┐
                ⋮                                            │ <2>
                ORG     Table start+(2*(Table area/4H)       ┘
```

[Description]

<1> Symbols are defined.
<2> The start address of the table area is defined in program memory.

(5) Remaindering operator (MOD)

[Format]

<expression-1>ΔMODΔ<expression-2>

[Function]

This operator takes the remainder resulting from dividing the value of <expression-1> by that of <expression-2>.

[Explanation]

If <expression-2> is equal to 0, the result of the operation is 0.

[Example]

```
Constant 1      DAT      552H

Constant 2      DAT      7H

R1              MEM      0.10H
                 ⋮
                ADD      R1,#Constant 1 MOD Constant 2
```

[Description]

In the above example, the result of Constant 1 MOD Constant 2 is 4H.

4.7.4 Logic Operators

(1) OR operator

[Format]

<expression-1>ΔORΔ<expression-2>

[Function]

This operator takes the OR of the values of <expression-1> and <expression-2>.

[Explanation]

A numeric value with a minus sign is processed as a 2s complement; the sign bit is processed as part of the numeric value.

[Example]

```
R1              MEM     1.40H

Constant 1      DAT     4H
                ⋮
                SUB     R1,#Constant 1 OR 8H
```

[Description]

In the above example, the result of Constant 1 OR 8H is 0CH.

(2) AND operator

[Format]

<expression-1>ΔANDΔ<expression-2>

[Function]

This operator takes the AND of the values of <expression-1> and <expression-2>.

[Explanation]

A numeric value with a minus sign is processed as a 2s complement; the sign bit is processed as part of the numeric value.

[Example]

```
Constant 1      DAT      4567H

R10             MEM      2.50H
                 ⋮
                MOV      R10,#(Constant 1/2H) AND 0FH
```

[Description]

In the above example, the result of (Constant 1/2H) AND 0FH is 03H. AND 0FH is used to validate only the lower four bits of a data type expression.

In the above example, an error occurs if AND 0FH is omitted because the range of values supported for the operand is exceeded.

(3) XOR operator

[Format]

<expression-1>ΔXORΔ<expression-2>

[Function]

This operator takes the exclusive OR of the values of <expression-1> and <expression-2>.

[Explanation]

A numeric value with a minus sign is processed as a 2s complement; the sign bit is processed as part of the numeric value.

[Example]

```
Constant A      DAT     2345H

Constant B      DAT     42H

R02             MEM     0.42H
                ⋮
                MOV     R02,#((Constant A–Constant B) XOR 0FH) AND 0FH
```

[Description]

In the above example, the result of ((Constant A–Constant B) XOR 0FH) AND 0FH is 0CH.
AND 0FH is used to validate only the lower four bits of a data type expression.
In the above example, an error occurs if AND 0FH is omitted because the range of values supported for the operand is exceeded.

(4) NOT operator

[Format]

NOT△<expression>

[Function]

This operator takes the 1s complement of the value of <expression>.

[Explanation]

A numeric value with a minus sign is processed as a 2s complement; the sign bit is processed as part of the numeric value.

[Example]

```
Constant        DAT     4567H

R9              MEM     0.12H

                ⋮

                MOV     R9,#(NOT Constant) AND 0FH
```

[Description]

In the above example, the result of (NOT Constant) AND 0FH is 8H.
AND 0FH is used to validate only the low-order four bits of a data type expression.
In the above example, an error occurs if AND 0FH is omitted because the range of values supported for the operand is exceeded.

4.7.5 Relational Operators

A relational operator compares the values to its right and left, returning –1 if the result is true or 0 if it is false.

(1) EQ (Equal) operator

[Format]

<expression-1>ΔEQΔ<expression-2>
or <expression-1>=<expression-2>

[Function]

This operator returns –1 (true) if the value of <expression-1> is equal to that of <expression-2>; otherwise, it returns 0 (false).

[Explanation]

EQ can be replaced with =.
–1 is processed as a 2s complement, thus is represented by 0FFFFFFFFH in hexadecimal.

[Example]

```
Condition       DAT     0AH                     <1>
R1              MEM     0.43H
                 :
Macro           MACRO   P1,P2,P3
                 IF     P1 EQ Condition
                 MOV    R1,#P2
                 :
                 ELSE                           <2>
                 MOV    R1,#P3
                 :
                 ENDIF
                ENDM
```

[Description]

<1> Symbols are defined.

<2> A macro is defined. P1, P2, and P3 are formal parameters.
If P1 = Condition, the statements between IF and ELSE are expanded. If P1 ≠ Condition, the statements between ELSE and ENDIF are expanded.

(2) NE (Not Equal) operator

[Format]

<expression-1>ΔNEΔ<expression-2>
or <expression-1><><expression-2>

[Function]

This operator returns –1 (true) if the value of <expression-1> is not equal to that of <expression-2>; otherwise, it returns 0 (false).

[Explanation]

NE can be replaced with <>.
–1 is processed as a 2s complement, thus is represented by 0FFFFFFFFH in hexadecimal.

[Example]

```
Condition       DAT     0BH                     ┐ <1>
R3              MEM     1.34H                   ┘
                 ⋮
Macro           MACRO   P1,P2,P3                ┐
                 IF     P1 NE Condition         │
                 MOV    R3,#P2                  │
                 ⋮                              │
                 ELSE                           │ <2>
                 MOV    R3,#P3                  │
                 ⋮                              │
                 ENDIF                          │
                ENDM                            ┘
```

[Description]

<1> Symbols are defined.

<2> A macro is defined. P1, P2, and P3 are formal parameters.
If P1 ≠ Condition, the statements between IF and ELSE are expanded. If P1 = Condition, the statements between ELSE and ENDIF are expanded.

(3) LT (Less Than) operator

[Format]

<expression-1>ΔLTΔ<expression-2>
or <expression-1><<expression-2>

[Function]

This operator returns –1 (true) if the value of <expression-1> is less than that of <expression-2>. If the value of <expression-1> is larger than or equal to that of <expression-2>, it returns 0 (false).

[Explanation]

LT can be replaced with <.
–1 is processed as a 2s complement, thus is represented by 0FFFFFFFFH in hexadecimal.

[Example]

```
Condition       DAT     02H                     <1>
R3              MEM     3.45H
                 :
Macro           MACRO   P1,P2,P3                <2>
                 IF     P1 LT Condition
                 MOV    R3,#P2
                 :
                 ELSE
                 MOV    R3,#P3
                 :
                 ENDIF
                ENDM
```

[Description]

<1> Symbols are defined.

<2> A macro is defined. P1, P2, and P3 are formal parameters.
If P1 < Condition, the statements between IF and ELSE are expanded. If P1 ≥ Condition, the statements between ELSE and ENDIF are expanded.

(4) LE (Less Than or Equal) operator

[Format]

<expression-1>ΔLEΔ<expression-2>
or <expression-1><=<expression-2>

[Function]

This operator returns –1 (true) if the value of <expression-1> is less than or equal to that of <expression-2>. If the value of <expression-1> is larger than that of <expression-2>, it returns 0 (false).

[Explanation]

LE can be replaced with <=.
–1 is processed as a 2s complement, thus is represented by 0FFFFFFFFH in hexadecimal.

[Example]

```
Condition       DAT     04H                     <1>
R1              MEM     1.13H
                 :
Macro           MACRO   P1,P2,P3                <2>
                 IF     P1 LE Condition
                 MOV    R1,#P2
                 :
                 ELSE
                 MOV    R1,#P3
                 :
                 ENDIF
                ENDM
```

[Description]

<1> Symbols are defined.

<2> A macro is defined. P1, P2, and P3 are formal parameters.
If P1 ≤ Condition, the statements between IF and ELSE are expanded. If P1 > Condition, the statements between ELSE and ENDIF are expanded.

(5) GT (Greater Than) operator

[Format]

<expression-1>ΔGTΔ<expression-2>
or <expression-1>><expression-2>

[Function]

This operator returns –1 (true) if the value of <expression-1> is greater than that of <expression-2>. If the value of <expression-1> is less than or equal to that of <expression-2>, it returns 0 (false).

[Explanation]

GT can be replaced with >.
–1 is processed as a 2s complement, thus is represented by 0FFFFFFFFH in hexadecimal.

[Example]

```
Condition       DAT     07H                    ┐
R1              MEM     3.44H                  ┘ <1>
                 :
Macro           MACRO   P1,P2,P3               ┐
                 IF     P1 GT Condition        │
                 MOV    R1,#P2                 │
                 :                             │
                 ELSE                          │ <2>
                 MOV    R1,#P3                 │
                 :                             │
                 ENDIF                         │
                ENDM                           ┘
```

[Description]

<1> Symbols are defined.

<2> A macro is defined. P1, P2, and P3 are formal parameters.
If P1 > Condition, the statements between IF and ELSE are expanded. If P1 ≤ Condition, the statements between ELSE and ENDIF are expanded.

(6) GE (Greater or Equal) operator

[Format]

<expression-1>ΔGEΔ<expression-2>
or <expression-1>>=<expression-2>

[Function]

This operator returns –1 (true) if the value of <expression-1> is greater than or equal to that of <expression-2>. If the value of <expression-1> is less than that of <expression-2>, it returns 0 (false).

[Explanation]

GE can be replaced with >=.
–1 is processed as a 2s complement, thus is represented by 0FFFFFFFFH in hexadecimal.

[Example]

```
Condition       DAT     0FH                     <1>
R1              MEM     1.67H
                 :
Macro           MACRO   P1,P2,P3                <2>
                 IF     P1 GE Condition
                 MOV    R1,#P2
                 :
                 ELSE
                 MOV    R1,#P3
                 :
                 ENDIF
                ENDM
```

[Description]

<1> Symbols are defined.

<2> A macro is defined. P1, P2, and P3 are formal parameters.
If P1 ≥ Condition, the statements between IF and ELSE are expanded. If P1 < Condition, the statements between ELSE and ENDIF are expanded.

4.7.6 Shift Operators

(1) SHR (Shift Right) operator

[Format]

<expression-1>ΔSHRΔ<expression-2>

[Function]

This operator shifts the bits of the value of <expression-1> to the right by a number of positions equal to the value of <expression-2>.

[Explanation]

The maximum allowable number of bits is 32. As a result of the shift, the MSB is set to 0.

[Example]

```
    Constant        DAT     00004578H

    Memory 1        MEM     0.48H
    Memory 2        MEM     0.49H
    Memory 3        MEM     0.4AH
    Memory 4        MEM     0.4BH
                     ⋮
                    MOV     Memory 1,#Constant AND 0FH
                    MOV     Memory 2,#Constant SHR 4 AND 0FH
                    MOV     Memory 3,#Constant SHR 8 AND 0FH
                    MOV     Memory 4,#Constant SHR 0CH AND 0FH
```

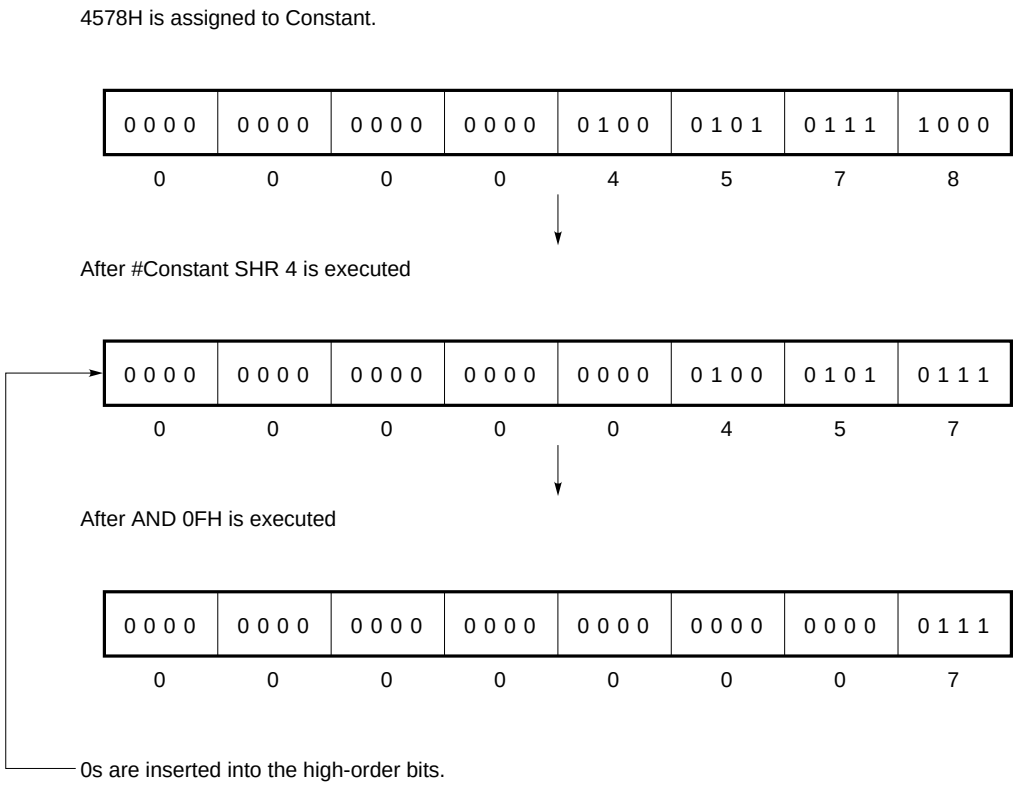
[Description]

In the above example, the numeric value assigned to symbol Constant is stored in 0.48H through 0.4BH in data memory.

AND 0FH is used to make only the lower four bits valid in a data type expression.

In the above example, an error occurs if AND 0FH is omitted because the range of values supported for the operand is exceeded.

The processing procedure for Memory 1, #Constant SHR 4 AND 0FH in the above example is as follows:



(2) SHL (Shift Left) operator

[Format]

<expression-1>ΔSHLΔ<expression-2>

[Function]

This operator shifts the bits of the value of <expression-1> to the left by a number of positions equal to the value of <expression-2>.

[Explanation]

The maximum allowable number of bits is 32. As a result of the shift, the LSB is set to 0.

[Example]

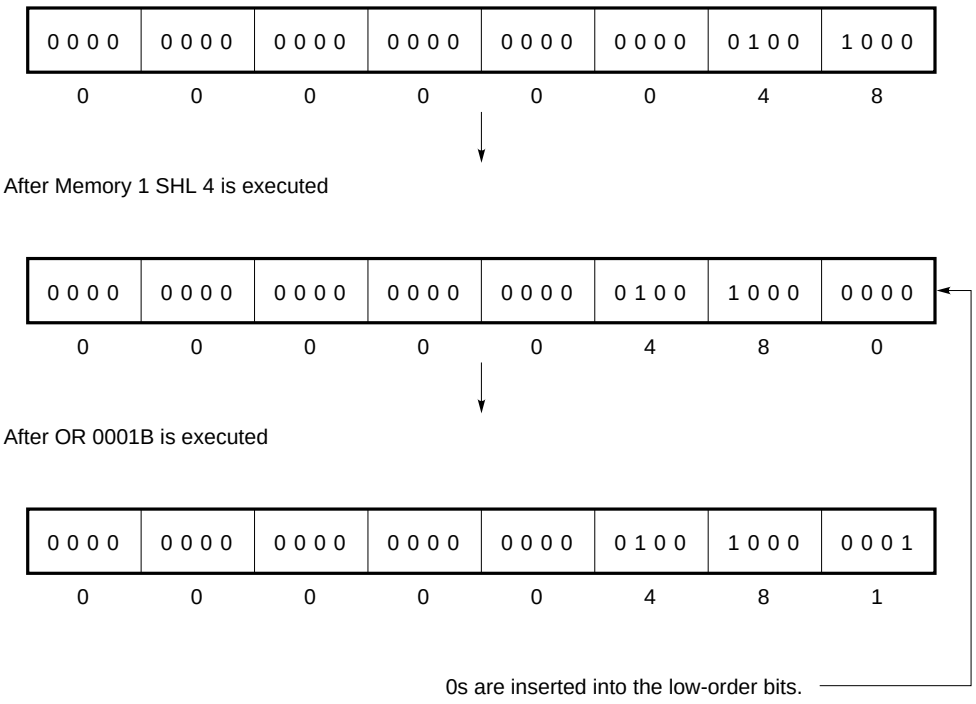
```
    Memory 1        MEM      0.48H
                      ⋮
                    SET1     .FM.Memory 1 SHL 4 OR 0001B
                      ⋮
```

[Description]

In the above example, a symbol defined as a data memory type symbol is used to set only one bit in the data memory specified by the symbol (the LSB of memory 1). SET1 is a built-in macro instruction that sets the flag at the position specified in the operand field. .FM. is a type conversion function that converts a symbol from memory type to flag type.

The processing procedure for .FM.Memory 1 SHL 4 OR 0001B in the above example is as follows:

The value of 0.48H is assigned to memory 1. First, memory 1 is converted from data memory type to flag type. At this time, the value of 0.48H is not affected.



4.7.7 () (Operation Order Specification Symbols)

[Format]

```
<expression-1> Operator (<expression-2> Operator <expression-3>)
(<expression-1> Operator <expression-2>) Operator <expression-3>
```

[Function]

A pair of these operators causes the operations enclosed in the parentheses to be performed first, irrespective of the operator priorities.

[Explanation]

If more than one pair of () has been specified, the operations enclosed in the innermost pair are performed first.

There is no limit on the number of () pairs that can be specified. However, one line can consist of no more than 255 characters.

[Example]

```
Constant 1      DAT     4789H
Constant 2      DAT     3H
Memory 1        MEM     0.48H
                 ⋮
                MOV     Memory 1,#((Constant 1 + Constant 2)*04H) AND 0FH
                 ⋮
```

[Description]

In the above example, () are used to perform addition processing Constant 1+Constant 2 before multiplication processing Constant 2*04H.

[MEMO]

CHAPTER 5 CONTROL SYMBOLS

5.1 EPA BIT CONTROL SYMBOLS (@AR_EPA0 AND @AR_EPA1)

These control symbols are valid in absolute mode only. They cannot be used in relocatable mode. This is because, in relocatable mode, the linker (LK17K) performs relocation section by section, such that the user cannot determine whether the location to which a jump occurs is in ROM or the EPA area. For this reason, RA17K supports the CALLX and BRX extended instructions. CALLX and BRX automatically control the EPA bit and are therefore easier to use from the standpoint of compatibility with AS17K.

Usually, the EPA bit is automatically controlled for a direct branch instruction (BR addr) and subroutine call instruction (CALL addr). That is, no matter where the program is located, the EPA bit is set if a branch to the EPA area occurs and reset if a branch to an area that can be created by the PC occurs.

If the operand for the branch instruction or the CALL instruction is an indirect specification (@AR), the assembler cannot determine the address of the location to which a branch occurs. If, therefore, BR @AR or CALL @AR is itself in the EPA area, a branch must occur within the EPA area; if it is in the user area, a branch must occur within the user area.

It can be seen, therefore, that the EPA area cannot be used intentionally during debugging. For example, the address of a location in the user area cannot be specified as the operand for an indirect branch instruction located in the EPA area.

To overcome this problem, the @AR_EPA0 and @AR_EPA1 control symbols are provided.

```
 E   STNO    LOC.      OBJ.     M  I    SOURCE STATEMENT

             00112     1D772             MOV   AR0, #.DL.LAB0 AND 0FH
             00113     1D760             MOV   AR1, #.DL.LAB0 SHR 4H AND 0FH
             00114     1D750             MOV   AR2, #.DL.LAB0 SHR 8H AND 0FH
             00115     1D748             MOV   AR3, #.DL.LAB0 SHR 12H AND 0FH
             00116     07040             BR    @AR

                                        ; EPA area
 1           10000     07090              INC   AR
 1           10001     07050              CALL  @AR
                                       LAB0:
                                            :
```

In the above example, BR @AR causes a branch to the EPA area. Therefore, to instigate a branch from location 00116H in the user area, the EPA bit must be set. To do this, the instruction at location 00116H must be changed from BR @AR to BR @AR_EPA1.

[Format]

EPA bit control symbols

```
@AR_EPA1 ;Sets the EPA bit and indicates the address indicated by @AR.
@AR_EPA0 ;Resets the EPA bit and indicates the address indicated by @AR.
```

Caution To enable the EPA bit control symbols, assemble-time variable ZZZEPA must be ≠ 0. The ZZZEPA assemble-time variable controls the enabling and disabling of @AR_EPA1 and @AR_EPA0.

```
ZZZEPA = 0  ; The EPA bit control symbols are disabled.
            ; The object code is the same as that when @AR is
            ; specified.

ZZZEPA ≠ 0  ; The EPA bit control symbols are enabled.
            ; The EPA bit is set and reset according to @AR_EPA1
            ; and @AR_EPA0.
```

[Function]

Symbol @AR_EPA1 or @AR_EPA0 can be used instead of operand @AR for an indirect branch instruction. The ZZZEPA assemble-time variable controls the enabling and disabling of @AR_EPA1 and @AR_EPA0. If ZZZEPA ≠ 0, the EPA bit is set by the statement containing @AR_EPA1 and reset by that containing @AR_EPA0.

[Example]

```
E   STNO    LOC.      OBJ.     M  I   SOURCE STATEMENT

                                      ; Manipulating indirect addresses
                                      ;
                      0001            ZZZEPA      SET   1
                                      ; The EPA bit control symbols are enabled.

                                      ; User area
            00001     07090           INC         AR
            00002     07050           CALL        @AR
            00003     07040           BR          @AR
1           00004     07040           BR          @AR_EPA1
                                      ; The EPA bit is set.
            00005     07040           BR          @AR_EPA0
                                      ; The EPA bit is reset.

                                      ; EPA area
1           10000     07090           INC         AR
1           10001     07050           CALL        @AR
1           10002     07050           CALL        @AR_EPA1
                                      ; The EPA bit is set.
            10003     07050           CALL        @AR_EPA0
                                      ; The EPA bit is reset.
```

[Notes]

(1) Use the SET pseudo instruction to change the value of ZZZEPA. The default is 0; @AR_EPA1 and @AR_EPA0 are disabled.

(2) The scope of ZZZEPA is limited in one source module file only. ZZZEPA cannot be passed to another module. Upon starting to assemble each module, ZZZEPA is set to its default value of 0.

(3) @AR_EPA0 and @AR_EPA1 cannot be used as the operand of the MOVT instruction. Therefore, the MOVT instruction cannot be used to reference a table in the user area from the EPA area. Nor can it be used to reference a table in the EPA area from the user area.

[MEMO]

CHAPTER 6 FUNCTIONS

RA17K supports functions previously defined by the system. Users cannot define functions.

Functions can be elements of an expression. They return predetermined values. The functions supported by RA17K are as follows:

- Type conversion function (temporarily changes the type of a symbol)
- $ ((location counter function) returns the value of the location counter)
- .TYPE. function (returns the type of a symbol)
- .DEF. function (returns a reference direction)
- .EV. function (returns the value of an environmental variable)
- ZZZLINE function (returns a line number in a source module file)
- ZZZARGC function (returns the number of parameters in a macro call statement)
- ZZZDEVID function (returns a device code)

[Notes]

(1) If an error occurs in a statement having a function, either of the following occurs:

- **If the function is written as the operand of an instruction for which object code will be created**
  The object code for an NOP instruction is created.

- **If the function is written as the operand of a pseudo instruction (no object code is created)**
  The pseudo instruction is nullified.

(2) Functions must be written as the operands of mnemonics and pseudo instructions. If a function is written by itself, an error (F037: Syntax error) occurs. No object code is created.

(3) Functions cannot be declared as PUBLIC. Otherwise, an error (F037: Syntax error) occurs. For an explanation of error handling, see **Section 12.1**.

(4) Functions cannot be declared as EXTRN. Otherwise, an error (F037: Syntax error) occurs and the EXTRN declaration is nullified. At the same time, all symbols in the EXTRN declaration are also nullified.

6.1 TYPE CONVERSION FUNCTION

RA17K does not permit the direct writing of data as the operand of an instruction or a pseudo instruction.

If a symbol is used to represent the operands of multiple instructions, the symbol types required for the operands of the individual instructions may differ.

If different types of the same value are to be used, defining different symbols for the respective types makes the program less readable. In addition, to change the value, all these symbols must be changed. To overcome this problem, RA17K supports symbol type conversion, allowing one symbol to be used for different types.

[Format]

```
.<target-type><current-type>.<expression>
```

[Function]

This function converts the type of the symbol written in <expression> to <target-type>.

[Explanation]

<expression> is converted to the target type according to the above conversion rule.
The types that can be specified in < > are as follows:

D : Data type
L : Label type
M : Data memory type
F : Flag type

Symbol type conversion by the type conversion function is effective only at the position where the function is written; the conversion is temporary.

[Notes]

(1) The data type is represented by D, irrespective of the data length.

(2) The data memory type is represented by M, irrespective of the number of nibbles. The evaluation value of a data memory type symbol is a defined value, plus nibble information; therefore, be careful when converting the type. (See **Section 8.5**.)

(3) For the flag type, the evaluation values of 0, 1, 2, and 3, indicated at the bit positions, are 1, 2, 4, and 8H. (See **Section 8.6**.)

(4) Changing a symbol from data memory type to flag type can be accomplished easily using . (period), as follows:

```
<data-memory-type-symbol-name>.<bit-position>

MEMORY    MEM     0.00H
          SET1    MEMORY.3     ; The MSB of MEMORY is set.
```

There is no need to use the type conversion function. If a value of 4 or more is written into the bit positions, an error (F044: Invalid value) occurs.

(5) Do not insert a blank (space or TAB) between . (period) and <target-type>, or between <current-type> and . (period). Otherwise, the description is not interpreted as a type conversion function and an error (F037: Syntax error) occurs. A blank may be inserted between . (period) and <expression>.

(6) If <current-type> and <expression> differ in type, an error (F045: Invalid type) occurs.

(7) If an undefined or forward reference symbol is written in <expression>, an error (F058: Undefined symbol) occurs.

(8) If no symbol name is written in <expression>, or if a character string (reserved word) other than a symbol name is written, an error (F037: Syntax error) occurs.

(9) A symbol declared as EXTRN can be written in <expression>.

(10) If the target type is the same as the current type, an error (F037: Syntax error) occurs.

Descriptions that cause errors

```
.DD.
.MM.
.FF.
.LL.
```

[Example]

(1) Examples with symbols written in <expression>

<1> Convert from data type to label type.

```
DATA1    DAT    0100H
LABEL1   LAB    .LD.DATA1
```

<2> Convert from data type to data memory type.

```
DATA2    DAT    0011H
MEMORY   MEM    .MD.DATA2
```

<3> Convert from data type to flag type.

```
DATA3    DAT    0022H
FLAG     FLG    .FD.DATA3
```

<4> Convert from label type to data type.

```
LABEL2   LAB    0200H
DATA1    DAT    .DL.LABEL2
```

<5> Convert from label type to data memory type.

```
LABEL4   LAB    0030H
MEMORY2  MEM    .ML.LABEL4
```

<6> Convert from label type to flag type.

```
LABEL3   LAB    0011H
FLAG2    FLG    .FL.LABEL3
```

<7> Convert from data memory type to data type.

```
MEMORY1  MEM    0.00H
DATA2    DAT    .DM.MEMORY1
```

<8> Convert from data memory type to label type.

```
MEMORY   MEM    1.00H
LABEL2   LAB    .LM.MEMORY
```

<9> Convert from data memory type to flag type.

```
MEMORY3  MEM    0.04H
FLAG3    FLG    .FM.MEMORY3
```

<10> Convert from flag type to data type.

```
FLAG     FLG   0.00.0H
DATA3    DAT   .DF.FLAG
```

<11> Convert from flag type to label type.

```
FLAG1    FLG   1.02.3H
LABEL3   LAB   .LF.FLAG1
```

<12> Convert from flag type to data memory type.

```
FLAG2    FLG   2.00.1H
MEMORY2  MEM   (.MF.FLAG2 SHR 4) AND 0FH
```

(2) Example with an operation expression in <expression> (data type to data memory type)

```
    ABCD    DAT  1000H
    .MD. (ABCD+1000H)
or  .MD.ABCD+1000H
```

(3) Example with an operation expression in <expression> (data memory type to data type)

```
    MEM1    MEM     1.00H
    MEM2    MEM     1.11H
    .DM. (MEM1+MEM2)
or  .DM.MEM1+.DM.MEM2
```

6.2 $ (LOCATION COUNTER FUNCTION)

[Format]

$

[Function]

In relocatable mode, this function returns the value of the offset location counter in the section.
In absolute mode, this function returns the current value (physical address) of the location counter.

[Explanation]

$ allows a relative address to be referenced easily.

[Notes]

(1) When $ is used, a warning (W114: The address carried out an operation using $ may be incorrect) is issued; object code is created, however.
If built-in macro instructions and user-defined macro instructions exist between the instruction containing $ and the address determined with the expression using the $, pay careful attention to the numbers of instructions into which these macro instructions will be expanded.

(2) $ returns the value of the label type.

[Example]

```
Memory   MEM    0.47H
           :
           :
          ADD    Memory,#01H
          SKT    Memory,#01H
          BR     $-2
```

In the above example, $–2 indicates (Current value of the location counter) – 2. By using $ in combination with operators, a relative address can be represented.

6.3 .TYPE. FUNCTION

[Format]

```
.TYPE.<expression>
```

[Function]

.TYPE. returns the type of the <expression> that follows immediately after it.
The following values are returned for the respective types:

Data type	(DAT) = 00H
Label type	(LAB) = 01H
Flag type	(FLG) = 02H
Data memory type	(MEM/NIBBLE/NIBBLE1) = 03H
	(NIBBLE2) = 13H
	(NIBBLE3) = 23H
	(NIBBLE4) = 33H
	(NIBBLE5) = 43H
	(NIBBLE6) = 53H
	(NIBBLE7) = 63H
	(NIBBLE8) = 73H
	(NIBBLE2V) = 93H
	(NIBBLE3V) = 0A3H
	(NIBBLE4V) = 0B3H
	(NIBBLE5V) = 0C3H
	(NIBBLE6V) = 0D3H
	(NIBBLE7V) = 0E3H
	(NIBBLE8V) = 0F3H

[Notes]

(1) A blank (space or TAB) may be inserted between .TYPE. and <expression>. Note, however, that inserting a blank between a period and TYPE or between TYPE and a period causes an error (F037: Syntax error) to occur.

(2) If the symbol specified in <expression> is a forward reference or undefined symbol, an error (F058: Undefined symbol) occurs and 0 is returned.

(3) If a symbol defined with a symbol definition pseudo instruction or a non-label type symbol (segment name, section name, table name) is written in <expression>, an error (F045: Invalid type) occurs and 0 is returned.

(4) Symbols representing the data, data memory, flag, and label types can be written, even if they are declared as EXTRN.

(5) If <expression> is not specified or a character string (reserved word) other than a symbol name is written in it, an error (F037: Syntax error) occurs and 0 is returned.

(6) .TYPE. returns the value of the data type.

(7) .TYPE. can be written in a macro. Also, a macro local symbol can be written in operand <expression>. (See **Section 16.4**.)

(8) If an operation expression of data memory type is written in <expression>, nibble information is determined from the operation result.

```
A   NIBBLE2   0.00H
B   SET       .TYPE. (A+A)
```

B is set to 23H because .TYPE. determines nibble information from the result of A+A.

(9) If an operation expression of label type is written in <expression>, 01H (label type) is always returned.

```
ENTRY       LAB : AAA
A     SET   AAA+123
```

A is set to 01H (label type).

[Example]

```
FLAG_TYPE   DAT    2
            :
            IF     .TYPE.SYM0 = FLAG_TYPE   ; If symbol SYM0 is flag type, the
            :                               ; IF clause will be expanded.
            ENDIF
```

6.4 .DEF. FUNCTION

[Format]

```
.DEF.<symbol-name>
```

[Function]

.DEF. returns the reference direction of the symbol specified by <symbol-name>, relative to the location where the function is written.
Reference can be performed either forwards or backwards. The meanings of the returned values are as follows:

0	:	Forward reference; or, the symbol is undefined.
–1 (0FFFFFFFFH)	:	Backward reference

The reference direction is based on the source module file. At the location where .DEF.<symbol-name> is written, if the symbol specified by <symbol-name> is already registered in a symbol table, –1 is returned; otherwise, 0 is returned.

[Notes]

(1) A blank (space or TAB) may be inserted between .DEF. and <symbol-name>. However, inserting a blank between a period and DEF, or between DEF and a period, causes an error (F037: Syntax error) to occur.

(2) If <symbol-name> is not specified or a character string (reserved word) other than a symbol name is written in it, an error (F037: Syntax error) occurs and 0 is returned.

(3) If a symbol declared as being EXTRN is written, 0 is returned.

(4) .DEF. returns the value of the data type.

(5) .DEF. returns the value for backward reference if the symbol specified in <symbol-name> has already been registered in a symbol table. When .DEF. is used in a macro, if the symbol is registered in either a macro local symbol table or GLOBAL symbol table, .DEF. returns the value for backward reference.

(6) Any user-defined symbol can be specified in <symbol-name>, such as those symbols defined with the DAT, FLG, SET, and MEM pseudo instructions and macro names.

(7) If .DEF. is written as the label of a CASE statement, it is regarding as being a numeric label such that an error (F069: Invalid CASE LABEL) can occur.

[Example]

(1) Symbol found with backward reference

```
SYM1    SET     1
        :

IF      .DEF.SYM1
        ; This part will be assembled because SYM1 was defined earlier.
        :

ELSE
        ; This part will not be assembled.
        :
ENDIF
```

(2) Undefined symbol

```
IF      .DEF.SYM2
        ; This part will not be assembled.
        :
ELSE
        ; This part will be assembled because SYM2 is not defined anywhere.
        :
ENDIF
```

(3) Symbol defined in another section

```
SEC1    CSEG
        :
SYM2    SET     1
        :

SEC2    CSEG
        :

IF      .DEF. SYM2
        ; This part will be assembled because SYM2 is defined in SEC1, which
        ; exists in the same module.
        :
ELSE
        :
ENDIF
```

(4) Symbol defined in another module

```
EXTRN   DAT:SYM2

IF      .DEF. SYM2
        :
ELSE
        ; This part will be assembled because SYM2 is defined in another
        ; module.
        :
ENDIF
```

6.5 .EV. FUNCTION

[Format]

```
.EV.<environmental-variable-name>
```

[Function]

This function returns the value of the environmental variable specified by <environmental-variable-name>. If the value of the environmental variable cannot be converted to a numeric value, –1 (0FFFFFFFFH) is returned.

[Notes]

(1) If the numeric value to which the value of the variable is converted exceeds the range of values that can be represented with 32 bits, the excess portion is ignored.

(2) If the environmental variable specified by <environmental-variable-name> is not found, or if its value cannot be converted to a numeric value, –1 is returned.

(3) A blank (space or TAB) may be inserted between .EV. and <environmental-variable-name>. Note, however, that inserting a blank between a period and EV, or between EV and a period, causes an error (F037: Syntax error) to occur.

(4) If <environmental-variable-name> is not specified, an error (F037: Syntax error) occurs.

(5) Any character string (including reserved words, but excluding special characters) can be written in <environmental-variable-name>. If special characters are used, an error (F037: Syntax error) occurs. –1 is returned.

(6) .EV. returns the value of the data type.

(7) Do not enclose <environmental-variable-name> in quotation marks. Otherwise, an error (F037: Syntax error) occurs and –1 is returned.

(8) The character string in <environmental-variable-name> must be entirely in upper case. If a character string is written in lower case, it is not converted to upper case. In this case, therefore, a value of –1 is always returned (an error does not occur).

[Example]

(1) To extract the value of environmental variable COUNTRY

```
IF      .EV.COUNTRY = 1           ; Environmental variable (COUNTRY)
    ;
    ZZZERROR    'WAIT A MINUTE!!'
    ZZZERROR    'PROGRAM CANNOT STOP RAPIDRY···'
    ;
ELSE
    ;
    ZZZERROR    'The program will not run as the user desires.'
    ;
ENDIF
```

(2) Variable whose value cannot be converted to a numeric value

```
IF      .EV.PATH<>-1              ; Environmental variable (PATH)
    ;
    ZZZMSG      'This system is in error.'
    ;
ELSE
    ;
    ZZZERROR    'Cannot converted to a numeric value.'
    ;
ENDIF
```

6.6 ZZZLINE FUNCTION

[Format]

```
ZZZLINE
```

[Function]

This function returns the line number of a statement in a source module file in a listing.
The same source line number is returned for the statements into which an include or macro instruction is expanded.

[Notes]

(1) If an operand is written for ZZZLINE, an error (F037: Syntax error) occurs.

(2) ZZZLINE returns the value of the data type.

[Example]

The following is an example of a macro that causes a warning message to be issued to the console on the line where the macro is written.

```
; Macro definition    " WARNING"

   WARNING               MACRO  LINE_NO
   ;
                         ZZZMSG     'WARNING AT LINE=&LINE_NO'
   ENDM
```

This macro is written to the location where a message would be issued, as follows:

```
   LINE      SET          ZZZLINE
   WARNING                %LINE      ; MACRO CALL
```

% in argument %LINE is an expression operator. % passes a 32-bit representation of the value of the <expression> immediately following it to a parameter. When this macro is expanded, the following message is displayed on the console:

```
  Module name (line number):WARNING AT LINE = 1234H
```

6.7 ZZZARGC FUNCTION

[Format]

ZZZARGC

[Function]

ZZZARGC, which can be written in a macro, returns the number of actual parameters in the macro in which it is written.

[Notes]

(1) Because ZZZLSARG is set to 0 by default, the macro will be expanded even if there are fewer actual parameters in the macro than the number of formal parameters. (See **Section 7.5**.)

(2) If this pseudo instruction is used outside a macro, an error (F145: Impossible to use out of macro) occurs. This line becomes invalid. At the same time, if it is written as the operand of an instruction for which object code will be created, the object code for an NOP instruction will be created; if it is written as the operand of a pseudo instruction, the pseudo instruction will be nullified.

(3) If the macro is nested, it will be as shown below.

```
LOC.   OBJ.   M  I    STATEMENT
                       :
              1        AMAC    P1,P2                    ; Expansion of AMAC
               :                :
              1        A       SET     ZZZARGC          ; A=2
               :                :
              2                BMAC    P1
               :                        :
              2                A       SET     ZZZARGC  ; A=1
               :                        :
              2                        ENDM
               :                :
              1        A       SET     ZZZARGC          ; A=2
               :                :
              1                ENDM
                       :
```

(4) If an operand is written for ZZZARGC, an error (F037: Syntax error) occurs.

(5) ZZZARGC returns the value of the data type.

[Example]

```
SSMAC   MACRO   X,Y,Z
        DW      X+Y
        ⋮
        IF      ZZZARGC=3
                DW  Z
        ENDIF
ENDM
```

If there are three actual parameters when the SSMAC macro is called, the IF block is expanded. If there are only two actual parameters, the IF block is not expanded.

```
SSMAC   61,62,63            ; Three parameters
        DW      61+62
        :
        IF      ZZZARGC=3
                DW      63  ; To be expanded.
        ENDIF

SSMAC   71,72               ; Two parameters
        DW      71+72
        :
        IF      ZZZARGC=3
                DW          ; Not to be expanded.
        ENDIF
```

6.8 ZZZDEVID FUNCTION

[Format]

```
ZZZDEVID
```

[Function]

This function returns the device number defined for a device file as a 32-bit value.

(1) If an operand is written for ZZZDEVID, an error (F037: Syntax error) occurs.

(2) ZZZDEVID returns the value of the data type.

Remark For details of device numbers, refer to the user's manual supplied with the device file corresponding to the target device.

[Example]

The following example shows a macro that supports the selection of an action according to the input device number because, in the library, the action differs according to the number of bits mounted in the address register.

```
TBL_JUMP    MACRO   DVID_PARAMETER, JUMP_ADDRESS
            ;
            CASE    DVID_PARAMETER
            ;
            03:                     ;AR=8 bits
                    MOV     AR0, #JUMP_ADDRESS AND 0FH
                    MOV     AR1, #JUMP_ADDRESS SHR 4 AND 0FH
                    BR      @AR
                    EXIT
            ;
            05:                     ;AR=12 bits
                    MOV     AR0, #JUMP_ADDRESS AND 0FH
                    MOV     AR1, #JUMP_ADDRESS SHR 4 AND 0FH
                    MOV     AR2, #JUMP_ADDRESS SHR 8 AND 0FH
                    BR      @AR
                    EXIT
            ;
            07:                     ;AR=14 bits
                    MOV     AR0, #JUMP_ADDRESS AND 0FH
                    MOV     AR1, #JUMP_ADDRESS SHR 4 AND 0FH
                    MOV     AR2, #JUMP_ADDRESS SHR 8 AND 0FH
                    MOV     AR3, #JUMP_ADDRESS SHR 12 AND 0FH
                    BR      @AR
                    EXIT
            ;
            OTHER:
                    ZZZERROR        'CANNOT USE THIS MACRO FOR THIS PRODUCT'
            ENDCASE
ENDM
```

The macro is referenced as follows:

```
            TBL_JUMP        ZZZDEVID , .DL.LABEL
```

The assembler reads a device code from the device file currently being used, selects the corresponding action, assigns the address (LABEL) of the location to which a branch occurs to the other parameter, then expands the macro.

CHAPTER 7 ASSEMBLE-TIME VARIABLES

RA17K supports variables that are unique to the system, and for which values can be set by both the assembler (system) and user-written SET pseudo instructions. These variables are called assemble-time variables.

An assemble-time variable is a symbol of data type. When referenced, it returns the value indicating the internal state of the system at that point, and so on. Also, by defining a value for an assemble-time variable with an instruction such as the SET pseudo instruction, the internal state of the system can be changed.

Assemble-time variables are registered as reserved words. The assemble-time variables are listed below:

- ZZZn (can be set to a value when assembly starts)
- ZZZSKIP (returns a value indicating whether the instruction immediately preceding the statement containing the variable has a skip function)
- ZZZBANK (returns the current bank number)
- ZZZPRINT (returns a list output state)
- ZZZLSARG (specifies whether an error occurs according to the number of parameters in a macro)
- ZZZSYDOC (controls the output of a symbol information table)
- ZZZALMAC (controls the output of the list associated with a user-defined macro)
- ZZZALBMAC (controls the output of the list associated with a built-in macro)
- ZZZEPA (controls the enabling and disabling of the EPA bit output control function)
- ZZZRP (returns the current value of the register pointer)
- ZZZAR (returns the current value of the address register)

[Notes]

(1) A value can be set for an assemble-time variable by using a SET pseudo instruction. If an <expression> of a type other than data type is written for the operand, an error (F045: Invalid type) occurs. This line becomes invalid. Therefore, the variable retains its previous value.

(2) An assemble-time variable is usually written in a parameter of a conditional assembly pseudo instruction. If an assemble-time variable is written by itself (either inside or outside a section block**Note**), an error (F037: Syntax error) occurs. This line becomes invalid. No object code is created.

Note A section block is a block defined by a CSEG pseudo instruction.

(3) If a parameter is written for an assemble-time variable, an error (F037: Syntax error) occurs. This line becomes invalid. At the same time, if the assemble-time variable is written as the operand of an instruction for which object code will be created, the object code for an NOP instruction will be created; if it is written as the operand of a pseudo instruction, the pseudo instruction will be nullified.

(4) An assemble-time variable returns a value of DAT type.

(5) An assemble-time variable cannot be declared as PUBLIC. If it is declared as PUBLIC, an error (F167: Invalid PUBLIC statement) occurs. For details of error handling, see **Section 12.1**.

(6) An assemble-time variable cannot be declared as EXTRN. If it is declared as EXTRN, an error (F166: Invalid EXTRN statement) occurs. For details of error handling, see **Section 12.2**.

7.1 ZZZn

[Function]

Optional switch ZZZn (where 0 ≤ n ≤ 15) is a variable for which a value can be set when assembly starts. As with ordinary options, to set a value, write -ZZZn=<numeric-value>.
In a program, ZZZn is used in the same way as a variable defined by a SET pseudo instruction. The only difference is that ZZZn is set to a value when assembly starts, provided the value is specified as an option.

[Notes]

(1) The range of <numeric-value> is from 0 to 0FFFFFFFFH (32 bits). If a numeric value that falls outside this range is written, an error (A106: Invalid option) occurs when assembly starts, such that assembly stops immediately.

(2) <numeric-value> can be specified in binary, decimal, or hexadecimal. If an expression or character string is specified for <numeric-value>, an error (A106: Invalid option) occurs when assembly starts, such that assembly stops immediately.

(3) The initial value of ZZZn (where 0 ≤ n ≤ 15) is 0 when assembly starts, unless a value is specified as an option.

(4) The value set for ZZZn in a module is valid within that module only.

(5) 0 cannot be inserted before n. For example, if ZZZ1 is written instead of ZZZ01, it is handled as an ordinary symbol, not as an assemble-time variable.

[Example]

In the following example, the user can specify whether to issue the error message by means of the ZZZERROR pseudo instruction when the assembler starts. If ZZZ9 ≠ –1, the error message is issued.

```
ZZZ_ERROR_MESSAGE_FOR_IRQ    MACRO
        IF (ZZZ9 <> –1) AND (ZZZIRQMES = 0)
            ZZZERROR 'CAUTION! Unexpected IRQ may be canceled.  See users manual'
        ENDIF
ENDM

SETIRQ1 MACRO    F1

        IF .DF. (F1 AND 0FE0H = 0BE0H)
                 ZZZ_ERROR_MESSAGE_FOR_IRQ
                   PEEK  WR,.MF.((F1) SHR 4)
                   OR    WR,#.DF.(F1) AND 0FH
                   ZZZPOKEIRQ     F1
        ELSE
                 SET1    F1
        ENDIF
ENDM
```

7.2 ZZZSKIP

[Function]

ZZZSKIP returns a value indicating whether the instruction for which object code will be created, written immediately before the statement in which ZZZSKIP is written, has a skip function. If the instruction has a skip function, ZZZSKIP returns a value of –1 (0FFFFFFFFH). Otherwise, 0 is returned.
Instructions that this variable recognizes as having a skip function are as follows:

(Machine language instructions)	SKE, SKNE, SKGE, SKLT, SKT, SKF
(Built-in macro instructions)	SKTn, SKFn, SKTX, SKFX, ADDSX, ADDCSX, SUBSX, SUBCSX, SKEX, SKNEX, SKGEX, SKGTX, SKLEX, SKLTX

[Notes]

(1) Even if the instruction for which object code will be created and which is written immediately before the statement containing ZZZSKIP has a skip function, ZZZSKIP returns 0 in the following cases:

- If the value of the location counter is changed with an ORG pseudo instruction, ZZZSKIP returns 0 even if the instruction written immediately before it has a skip function.
- In absolute mode, if the value of the location counter is changed by executing a CSEG pseudo instruction, ZZZSKIP returns 0, even if the instruction written immediately before it has a skip function.
- If a CSEG pseudo instruction is written immediately before the statement containing ZZZSKIP, ZZZSKIP returns 0 regardless of the instruction immediately preceding the CSEG pseudo instruction.

(2) The value of ZZZSKIP can be changed to another value by using a SET pseudo instruction.

(3) If the instruction immediately preceding the statement containing ZZZSKIP has a skip function, but the instruction has caused an assemble error, ZZZSKIP is set to 0. This is because the object code for a NOP instruction is created due to the occurrence of the error.

[Example]

In the following example, when the macro instruction is expanded, if the instruction for which object code will be created and which is written immediately before it has a skip function, branch instructions are automatically created to avoid logical conflicts. If ZZZSKIP returns 0, the instruction is deemed not to have a skip function. Otherwise, it is assumed to have a skip function. Thus, optimum object code will be created.

```
SETIRQ1         MACRO   F1
        IF      ZZZSKIP
          BR    $ + 2
          BR    $ + 4
        ENDIF
        PEEK    WR,.MF.((F1) SHR 4)
        OR      WR,#.DF.(F1) AND 0FH
        ZZZPOKEIRQ      F1
ENDM
;
```

7.3 ZZZBANK

[Function]

ZZZBANK returns the value indicating whether a built-in macro instruction (BANKn or SETBANK) or a label line appear in a statement later in the program. If a built-in macro instruction (BANKn or SETBANK) appears, ZZZBANK returns the bank number set by the built-in macro instruction; if a label line appears, –1 (0FFFFFFFFH) is returned.

[Notes]

(1) If a statement contains an instruction for manipulating a BANK register other than that of a built-in macro instruction (BANKn or SETBANK), information on the bank number is not stored in ZZZBANK.

(2) The value returned by ZZZBANK is valid in only the section block in which it is written. At the beginning of a section block, ZZZBANK is always initialized to –1.

(3) A value can be set for ZZZBANK by using a SET pseudo instruction. If an attempt is made to set a value that does not exist in the device, an error (F046: Invalid BANK No.) occurs and ZZZBANK retains the previous value.

[Example]

An example of an automatic bank switching macro is shown below:

```
BNKCHG     MACRO     AA
  IF     ZZZBANK = –1                    ; If a label line appears before a
         BANK&AA                         ; reference is made to this macro, bank
                                         ; switching is performed forcibly.
  ELSE
    IF       ZZZBANK <> AA SHR 8 AND 0FH  ; If ZZZBANK contains a value other
             BANK&AA                      ; than –1, ZZZBANK and parameter AA are
    ENDIF                                 ; compared to determine whether bank
  ENDIF                                   ; switching should be performed.
ENDM
```

By using the above macro in a program, whether bank switching should be performed in the program is automatically determined so that bank switching can be performed with a minimum number of instructions.

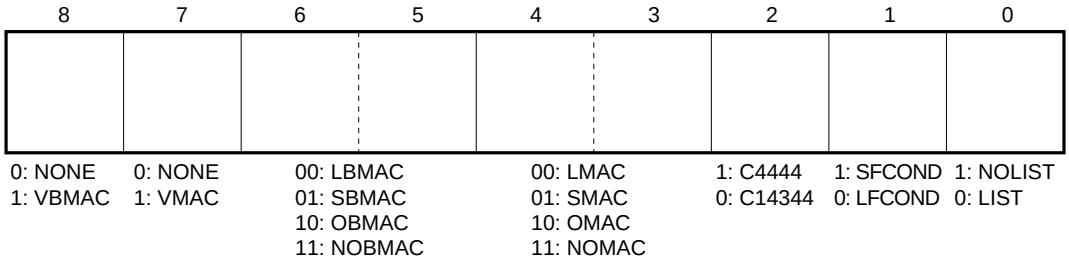
7.4 ZZZPRINT

[Function]

ZZZPRINT returns the assemble list output status specified immediately before a NOLIST pseudo instruction. The following list output control instruction states can be set in ZZZPRINT:

LIST/NOLIST
SMAC/VMAC/OMAC/NOMAC/LMAC
SBMAC/VBMAC/OBMAC/NOBMAC/LBMAC
SFCOND/LFCOND
C14344/C4444

Nine bits are set in ZZZPRINT. Each bit has the following meaning:



A list output status can also be set by changing the value of ZZZPRINT with a SET pseudo instruction. For example, if ZZZPRINT is set to 001001110B (04EH), OBMAC/SMAC/SFCOND/LIST take effect in the subsequent lines.

ZZZPRINT has a 32-bit evaluation value; however, the high-order 23 bits are fixed to 0s. Therefore, even if a value exceeding 9 bits is set in ZZZPRINT by using a SET pseudo instruction, the high-order 23 bits are cleared to 0s.

[Notes]

(1) ZZZPRINT is always reset to 0 when assembly starts.

(2) The value of ZZZPRINT is effective from the location of a NOLIST pseudo instruction to the point where another NOLIST pseudo instruction appears, unless the value is changed by using a SET pseudo instruction. Within this range, even if another list output control pseudo instruction exists, the value of ZZZPRINT does not change. Only when a NOLIST pseudo instruction appears is its information set in ZZZPRINT.

(3) If the value to be set in ZZZPRINT exceeds 9 bits, only the low-order 9 bits are set. No error occurs at this time.

(4) In the NOLIST state, if the state is changed to LIST by setting a value in ZZZPRINT, the statement that sets the value itself is not printed on the list.

```
SMAC
  :
NOLIST
        A       SET     ZZZPRINT    ; A value indicating the state SMAC/LIST
  :                                 ; is set in A.
  :
  :
        ZZZPRINT        SET     A   ; This line itself is not printed.
                                    ; SMAC/LIST take effect in the subsequent
                                    ; lines.
```

(List output)

[Example]

When a NOLIST control instruction is used in a macro, ZZZPRINT can be used after macro expansion to restore the list output control state to the state existing immediately before macro expansion.
In the following example, list output is prohibited by NOLIST in a macro during macro expansion.

```
LOC.  OBJ.  M  I  STATEMENT

                  DATA_SET          MACRO   MAX_VALUE,MEMA,FLGA,MEMB,FLGB
                            NOLIST                              ;SET ZZZPRINT
                            ;
                            SFCOND                              ;
                            IF      MAX_VALUE < 0FH
                                    LD      CHANGE_DATA,MEMA-1
                                    LD      CHANGE_DATA,MEMA
                                    MOV     DATA_MAX-1,#MAX_VALUE SHR 4
                                    MOV     DATA_MAX,#MAX_VALUE AND 0FH
                                    SET1    FLGA
                            ELSE
                                    LD      CHANGE_DATA,MEMB-1
                                    LD      CHANGE_DATA,MEMB
                                    MOV     DATA_MAX-1,#MAX_VALUE SHR 4
                                    MOV     DATA_MAX,#MAX_VALUE AND 0FH
                                    SET1    FLGB
                            ENDIF
                            LFCOND
                            LIST
                  ENDM
```

If the list output control state is NOLIST before reference is made to the DATA_SET macro, NOLIST is canceled by the last LIST pseudo instruction when this macro is expanded. After macro expansion, to restore the list output control state to the same state as that existing immediately before the macro reference, ZZZPRINT can be used as follows:

```
SFCOND
   ⋮
DATA_SET          0EH,MEMORY1,FLG1
          NOLIST                                ;SET ZZZPRINT
          LIST_CON        SET      ZZZPRINT     ;LIST_CON = 2H
          ;
          SFCOND
                  LD      CHANGE_DATA,MEMB–1
                  LD      CHANGE_DATA,MEMB
                  MOV     DATA_MAX–1,#MAX_VALUE SHR 4
                  MOV     DATA_MAX,#MAX_VALUE AND 0FH
                  SET1    FLGB
          LFCOND
ZZZPRINT          SET     LIST_CON ;After macro expansion, the
                                   ;state returns to SFCOND,
ENDM                               ;the state existing
                                   ;immediately before the
                                   ;reference.
```

7.5 ZZZLSARG

[Function]

ZZZLSARG is used to select the action to be performed if the number of actual parameters specified during macro expansion is smaller than the number of defined formal parameters.

ZZZLSARG ≠ 0 : If the number of actual parameters < number of formal parameters, an error (F036: Operand count error) occurs. The macro is not expanded.

ZZZLSARG = 0 : If the number of actual parameters < number of formal parameters, no error occurs. Macro parameters are passed in the order they appear, starting from the left.

[Notes]

(1) ZZZLSARG is always reset to 0 when assembly starts.

(2) Before using a ZZZARGC function, the value of ZZZLSARG must be 0. If ZZZLSARG ≠ 0 and the number of actual parameters < number of formal parameters, an error occurs. The macro is not expanded and the ZZZARGC function becomes invalid.

(3) If the number of actual parameters for a macro exceeds the number of formal parameters, an error (F036: Operand count error) occurs regardless of the value of ZZZLSARG. The macro is not expanded.

[Example]

```
MAC1    MACRO   P1,P2       ;Two parameters
          :
        ENDM
;
ZZZLSARG        SET -1
        MAC1    R1          ;An error occurs.
;
ZZZLSARG        SET 0
        MAC1    R2          ;No error occurs.
```

7.6 ZZZSYDOC

[Function]

ZZZSYDOC controls the output of the symbol information table by the document creation function.

ZZZSYDOC = 0 : The output of the symbol information table, created for each routine, is prohibited.
ZZZSYDOC ≠ 0 : The symbol information table created for each routine is output.

[Notes]

(1) The value of ZZZSYDOC is 1 when assembly starts. A symbol table is output. (Refer to the **Document Processor (DOC17K) User's Manual** for details.)

7.7 ZZZALMAC

[Function]

ZZZALMAC controls the output format of the list associated with the expansion of a user-defined macro (including libraries).

The intermediate list output by RA17K is not controlled by ZZZALMAC; the list output by the document processor (DOC17K) is controlled.

ZZZALMAC = 0 : Macro expansion starts from the same column as macro definition.

ZZZALMAC ≠ 0 : Macro expansion starts according to the macro call statement. This means that macro expansion starts from the same column as the first character of the macro name in the macro call statement.
If the macro call statement is indented, macro expansion is also indented by the same number of columns.

[Notes]

(1) ZZZALMAC is always reset to 0 when assembly starts.

(2) If the value of ZZZALMAC is changed in a macro, the value takes effect in the statements subsequent to ZZZALMAC in the macro.

[Example] Macro definition and lists output by DOC17K

```
LOC.  OBJ.  M  I   STATEMENT
                     MEM_CLR     MACRO    START , END
                              MOV     IXH , #.DM.START SHR 9 AND 01H
                              MOV     IXM , #.DM.START SHR 5 AND 08H
                              MOV     IXL , #00H
                              MOV     RG1 , #.DM.END SHR 9 AND 01H
                              MOV     RG2 , #.DM.END SHR 5 AND 08H
                              CALL    MEMCLR
                     ENDM

                     ZZZALMAC           SET    -1
                                     MEM_CLR          MEM1 , MEM2
               +                                MOV     IXH , #.DM.START SHR 9 AND 01H
               +                                MOV     IXM , #.DM.START SHR 5 AND 08H
               +                                MOV     IXL , #00H
               +                                MOV     RG1 , #.DM.END SHR 9 AND 01H
               +                                MOV     RG2 , #.DM.END SHR 5 AND 08H
               +                                CALL    MEMCLR

                     ZZZALMAC           SET     0
                              MEM_CLR          MEM1 , MEM2
               +              MOV     IXH , #.DM.START SHR 9 AND 01H
               +              MOV     IXM , #.DM.START SHR 5 AND 08H
               +              MOV     IXL , #00H
               +              MOV     RG1 , #.DM.END SHR 9 AND 01H
               +              MOV     RG2 , #.DM.END SHR 5 AND 08H
               +              CALL    MEMCLR
```

7.8 ZZZALBMAC

[Function]

ZZZALBMAC controls the output format of the list associated with the expansion of an RA17K built-in macro instruction.

The intermediate list output by RA17K is not controlled by ZZZALBMAC; the list output by the document processor (DOC17K) is controlled.

ZZZALBMAC = 0 : Macro expansion starts from the first column of the assembler list.

ZZZALBMAC ≠ 0 : Macro expansion starts according to the macro call statement. This means that macro expansion starts from the same column as the first character of the macro name in the macro call statement.
If the macro call statement is indented, the macro expansion is also indented by the same number of columns.

[Notes]

(1) ZZZALBMAC is always reset to 0 when assembly starts.

[Example] Macro description and lists output by DOC17K

```
LOC.  OBJ.  M  I  STATEMENT
                    MEMORY_F          FLG    0.20H.0
                    DATA_F            FLG    0.40H.0
                                      :
                                      :
                                      :
                    ZZZALBMAC         SET    -1

                            SET2    MEMORY_F,DATA_F
              +             OR      .MF.MEMORY_F SHR 4,#.DF.MEMORY_F AND 0FH
              +             OR      .MF.DATA_F SHR 4,#.DF.DATA_F AND 0FH

                    ZZZALBMAC         SET    0

                            CLR2    MEMORY_F , DATA_F
              +       AND     .MF.MEMORY_F SHR 4,#.DF.MEMORY_F AND 0FH
              +       AND     .MF.DATA_F SHR 4,#.DF.DATA_F AND 0FH
```

7.9 ZZZEPA

[Function]

Assemble-time variable ZZZEPA is valid in absolute mode only. This is because the control symbols @AR_EPA1 and @AR_EPA0 are valid in absolute mode only.

This variable specifies whether the EPA bit output control function (@AR_EPA0 and @AR_EPA1) is to be enabled or disabled.

ZZZEPA = 0 : Control symbols @AR_EPA1 and @AR_EPA0 are disabled.
ZZZEPA ≠ 0 : Control symbols @AR_EPA1 and @AR_EPA0 are enabled.

Whether the EPA bit output control function is to be enabled or disabled corresponds to whether the EPA bit should be manipulated by the statements in a program in which symbols @AR_EPA0 and @AR_EPA1 are written. If ZZZEPA ≠ 0, the EPA bit is cleared from the statement in which @AR_EPA0 is written, and set in the statement in which @AR_EPA1 is written. If ZZZEPA = 0, @AR_EPA1 and @AR_EPA0 have the same effect as @AR.

[Notes]

(1) ZZZEPA cannot be passed between modules. ZZZEPA is always reset to 0 at the start of module assembly. (At the beginning of each module, the EPA bit output control function is disabled.)

(2) If ZZZEPA is not written in a program, the assembler assumes that ZZZEPA is equal to 0, such that the EPA bit output control function is disabled.

(3) Using ZZZEPA in relocatable mode causes an error (F112: Impossible to use on relocatable mode).

[Example]

In the following example, symbols @AR_EPA1 and @AR_EPA0 are written instead of operand @AR for indirect branch instructions.

```
E   STNO    LOC.      OBJ.     M  I    SOURCE STATEMENT

                                       ; Manipulating indirect addresses
                                       ;
                      0001             ZZZEPA      SET   1
                                       ; The EPA bit control symbols are enabled.
            00001     07090            INC         AR
            00002     07050            CALL        @AR
1           00003     07040            BR          @AR_EPA1   ; EPA bit ON
            00004     07040            BR          @AR_EPA0   ; EPA bit OFF
                                       ;
                                       ORG         10000H     ; EPA AREA
1           10000     07090            INC         AR
            10001     07050            CALL        @AR_EPA0   ; EPA bit OFF
                      0001             ZZZEPA      SET   0
                                       ; The EPA bit control symbols are disabled.
1           10002     07050            CALL        @AR_EPA0   ; EPA bit ON
```

7.10 ZZZRP

[Function]

ZZZRP returns the value of the register pointer set by a built-in macro instruction (SETRP). If, however, a label exists between the SETRP and ZZZRP, –1 (0FFFFFFFFH) is returned.

[Notes]

(1) If the register pointer (RP) is set to a value without using a SETRP instruction, ZZZRP does not reflect that value.

(2) If a label exists between a SETRP instruction and ZZZRP, ZZZRP returns –1 (0FFFFFFFFH). The reason for this is that ZZZRP is set to –1 if a label exists between them because, during the execution of the program, the program may branch to the label without first passing through SETRP.

(3) ZZZRP is valid in only the section block in which it is written. At the beginning of a section block, ZZZRP is always set to –1.

(4) ZZZRP can be set to a value with a SET pseudo instruction. If an attempt is made to set it to a value that does not exist in the device, an error (F044: Invalid value) occurs and ZZZRP retains the previous value.

(5) If the register pointer (RP) is set to a value with an ENSURE pseudo instruction, ZZZRP reflects that value.

[Example]

Example of automatic register pointer switching macro

```
RPCHG           MACRO       NO      ;An MEM attribute symbol must have
                                    ;been previously specified for NO.
  IF    ZZZRP <> .DM.NO SHR 4 AND 0F7H
                SETRP   NO
  ENDIF
ENDM
```

7.11 ZZZAR

[Function]

ZZZAR returns the value of the register pointer set by a built-in macro instruction (SETAR). If, however, a label exists between the SETAR and ZZZAR, –1 (0FFFFFFFFH) is returned.

[Notes]

(1) If the register pointer (AR) is set to a value without using a SETAR instruction, ZZZAR does not reflect the value.

(2) If a label exists between a SETAR instruction and ZZZAR, ZZZAR returns –1 (0FFFFFFFFH). The reason for this is that ZZZAR is set to –1 if a label exists between them because, during the execution of the program, the program may branch to the label without first passing through SETAR.

(3) ZZZAR is valid only in the section block in which it is written. At the beginning of a section block, ZZZAR is always set to –1.

(4) ZZZAR can be set to a value with a SET pseudo instruction. If an attempt is made to set it to a value that does not exist in the device, an error (F044: Invalid value) occurs and ZZZAR retains the previous value.

(5) If the register pointer (AR) is set to a value with an ENSURE pseudo instruction, ZZZAR reflects the value.

[Example]

Example of automatic register pointer switching macro

```
ARCHG           MACRO             NO  ;An LAB attribute symbol must have
                                      ;been previously specified for NO.
  IF    ZZZAR <> .DL.NO
                SETAR   NO
  ENDIF
ENDM
```

CHAPTER 8 SYMBOL DEFINITION PSEUDO INSTRUCTIONS

This chapter explains the data to be described in programs and the types of labels and variables that reference storage locations (addresses) for instructions and data. It also explains how to declare the symbols.

The symbol definition pseudo instructions include the following:

- DAT pseudo instruction (define a data type name.)
- LAB pseudo instruction (define a label type name.)
- MEM pseudo instruction (define a memory type name.)
- NIBBLE pseudo instruction (define a memory type name.)
- NIBBLEn pseudo instruction (define a memory type name.)
- NIBBLEnV pseudo instruction (define a memory type name.)
- FLG pseudo instruction (define a flag type name.)
- SET pseudo instruction (define a name.)

[Basic format]

```
<name>Δsymbol-definition-pseudo-instructionΔ<expression>
```

[Notes]

(1) If an operand is omitted or two or more operands are described, an error (F037: Syntax error) occurs. <name> is registered as evaluation value 0.

(2) If <name> is omitted, an error (F037: Syntax error) occurs.

(3) If an undefined symbol or forward reference symbol is described in <expression>, an error (F058: Undefined symbol) occurs. <name> is registered as evaluation value 0.

(4) If the <expression> description is invalid, an error (see **Section 4.7.1** for details) occurs. <name> is registered as evaluation value 0.

(5) If a reserved word is described in <name>, an error (F037: Syntax error) occurs. However, upon assembly, the SET pseudo instruction can be used to set a value in a variable.

(6) If a defined symbol is not referenced, a warning (W020: Unreference symbol) is output. This warning, however, is output only to the log file. It is not output for a line in which a symbol that is not referenced on the intermediate list is defined.

(7) A defined symbol cannot be redefined in <name>. If it is redefined, an error (F057: Symbol multi defined) occurs. This line is invalidated. However, redefinition is possible for a local symbol in a macro.
A symbol defined with the SET pseudo instruction can be set again with the SET pseudo instruction.

(8) See **Section 4.3** for details of the <name> description.

(9) If the EXTRN symbol is described in operand <expression>, an error (F150: Impossible to write the external symbol) occurs. <name> is registered as evaluation value 0.

(10) If an error occurs in explanations (1) - (9), above, and <name> is registered as evaluation value 0, the following types are used:

DAT	:	Data type
LAB	:	Label type
MEM	:	Data memory type
NIBBLE	:	Data memory type
NIBBLEn	:	Data memory type
NIBBLEnV	:	Data memory type
FLG	:	Flag type
SET	:	The data type is retained for new registration. The previous type is retained when a registered symbol is set again. For this reset, the previous value is also retained.

8.1 SYMBOL DECLARATION

In RA17K, data and addresses used by instructions and pseudo instructions cannot be described only with numerics or numeric expressions. Before these numerics are used, they must be defined or declared as symbols. A symbol can be defined or declared at any point in a program. However, the scope is defined according to the declaration location.

Types are added for these symbols. The types of symbols used by certain instructions and pseudo instructions are predetermined; therefore, these types are useful for locating mistakes at program creation. Cross reference creation, memory map creation, and document creation by the document processor are accomplished by adding types to symbols.

With RA17K, a program can be split into two or more modules. A symbol is a local symbol that can be used only in a certain module and which cannot be referenced from other modules. To reference a symbol defined by an external module, the user must declare that the symbol is used by the external module (PUBLIC declaration).

The purpose of this definition is to enhance program development efficiency and program maintainability.

8.2 SYMBOL TYPES

RA17K defines symbol types for items (1) to (4), below:

(1) Data type: Constant, immediate data

```
DAT
```

(2) Label type: Address in program memory (ROM)

```
LAB
```

(3) Data memory type: Address in data memory (RAM) or register file (RF) (in nibbles)

```
MEM         NIBBLE
NIBBLE1
NIBBLE2     NIBBLE2V
:           :
:           :
:           :
NIBBLE8     NIBBLE8V
```

(4) Flag type: Flag in data memory (RAM) or register file (RF) (in bits)

```
FLG
```

To define a symbol, use a symbol definition pseudo instruction. In addition to (1) to (4), the SET pseudo instruction is used as the symbol definition pseudo instruction.

A value assigned by a symbol definition pseudo instruction cannot be changed. However, the value of a symbol defined by the SET pseudo instruction can be changed by using the SET pseudo instruction. Therefore, the SET pseudo instruction is used to define variables that are meaningful only at assembly. The assemble-time variables include user-defined variables defined within a program and assemble-time variables whose values can be assigned either by the assembler system or the SET pseudo instruction of the program.

The assemble-time variables are used as the parameters of macros and conditional pseudo instructions.

8.3 DAT PSEUDO INSTRUCTION

[Format]

<name>ΔDATΔ<expression>[Δ][;<comment>]

[Function]

Assigns the value of <expression>, described in the operand, to <name>. The evaluation value of <expression> is 32 bits.

[Explanation]

(1) A blank (space or TAB) must be inserted between <name> and the pseudo instruction, and also between the pseudo instruction and <expression>. If no blank is inserted, the line is not recognized as a DAT pseudo instruction and an error (F037: Syntax error) occurs. This line is invalidated. Therefore, the symbol described in <name> is not registered.

(2) If the pseudo instruction description is invalid, an error (F037: Syntax error) occurs. This line is invalidated. Therefore, the symbol described in <name> is not registered.

(3) Any operator other than the bit delimiting operator (.) can be used in <expression>.

(4) If an <expression> other than the data type is described, an error (F045: Invalid type) occurs. <name> is registered as evaluation value 0.

8.4 LAB PSEUDO INSTRUCTION

[Format]

<name>ΔLABΔ<expression>[Δ][;<comment>]

[Function]

Assigns the value of <expression>, described in the operand, to <name>.
<name> is used as a label type symbol.

[Explanation]

(1) A blank (space or TAB) must be inserted between <name> and the pseudo instruction, and also between the pseudo instruction and <expression>. If no blank is inserted, or if the description is invalid, the line is not recognized as a LAB pseudo instruction and an error (F037: Syntax error) occurs. This line is invalidated. Therefore, the symbol described in <name> is not registered.

(2) The evaluation value assigned by <expression> can consist of up to 32 bits.
If the evaluation value of <expression> is set in other than the ROM area, the following results:

- **If the evaluation value is set in the EPA area**
  A warning (W153: The address is in EPA area) is output.

- **If the evaluation value is set in other than the ROM or EPA areas**
  An error (F152: The address is out of ROM) occurs and <name> is not registered.

(3) Any operator other than the bit delimiting operator (.) can be used in <expression>.

(4) If <expression> other than a label or data type is described, an error (F045: Invalid type) occurs. <name> is registered as evaluation value 0.

[Notes]

(1) In general, a label is defined in a program by coding <symbol-name>: at the beginning of the line to which an address is to be assigned.

[Example]

```
TABLE1:    MOV    A, #1H
             :
           BR     TABLE1
```

The label type symbol definition pseudo instruction is used to reference an address of another source module file.

(2) To reference an entry address within the 17K series system segment by using the SYSCAL instruction, describe a data type symbol, not a label type symbol, in the operand.

(3) A value of <expression>, described in an operand of a label type symbol definition pseudo instruction, is used as an absolute address. The value of the symbol defined by <label>: is an offset address. However, the value of a symbol defined by the label type definition pseudo instruction is used as an absolute value.

8.5 MEM PSEUDO INSTRUCTION

With RA17K, a symbol name can be defined as a data memory type for two or more nibbles in data memory. The number of nibbles is specified by the pseudo instruction.

The following table lists the storage areas (number of nibbles) and value ranges to be declared by the pseudo instructions:

Type	Storage area	Value range
MEM	1 nibble	0 to 15
NIBBLE	1 nibble	0 to 15
NIBBLE1	1 nibble	0 to 15
NIBBLE2 (2V)	2 nibbles	0 to 255
NIBBLE3 (3V)	3 nibbles	0 to 4095
NIBBLE4 (4V)	4 nibbles	0 to 65535
NIBBLE5 (5V)	5 nibbles	0 to 1048575
NIBBLE6 (6V)	6 nibbles	0 to 16777215
NIBBLE7 (7V)	7 nibbles	0 to 268435455
NIBBLE8 (8V)	8 nibbles	0 to 4294967295

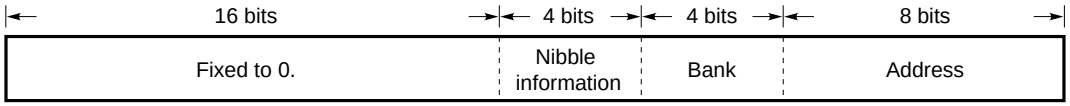
These types are collectively referred to as a data memory type.

The data memory type symbol definition pseudo instruction has the same name as that of the symbol type listed in the above table. All <expression> items described in pseudo instruction operations are of the same type and specify addresses in data memory.

When the type of a multi-nibble is declared, if it is Horizontal (= horizontal symbol) defined by the NIBBLEn instruction, the upper area, from the specified address to the column, is allocated. The allocated area can thus extend over two or more low-order addresses.

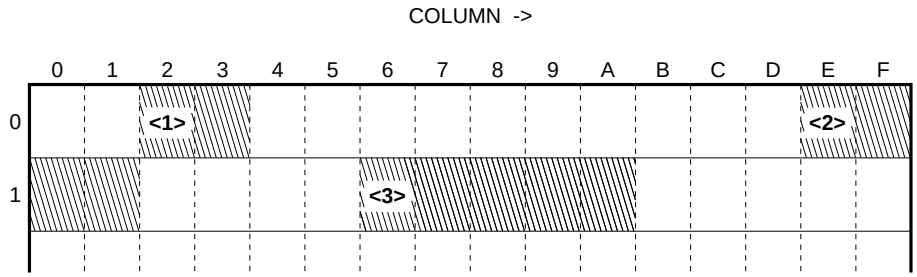
If the declared multi-nibble is Vertical (= vertical type) defined by the NIBBLEnV instruction, the upper addresses, from the specified address to the low-order address, are allocated. An error occurs if 7H of the low-order address is exceeded (see **[Explanation]** below).

The evaluation value indicated by the symbol is that obtained by adding nibble information to a value defined by the symbol.



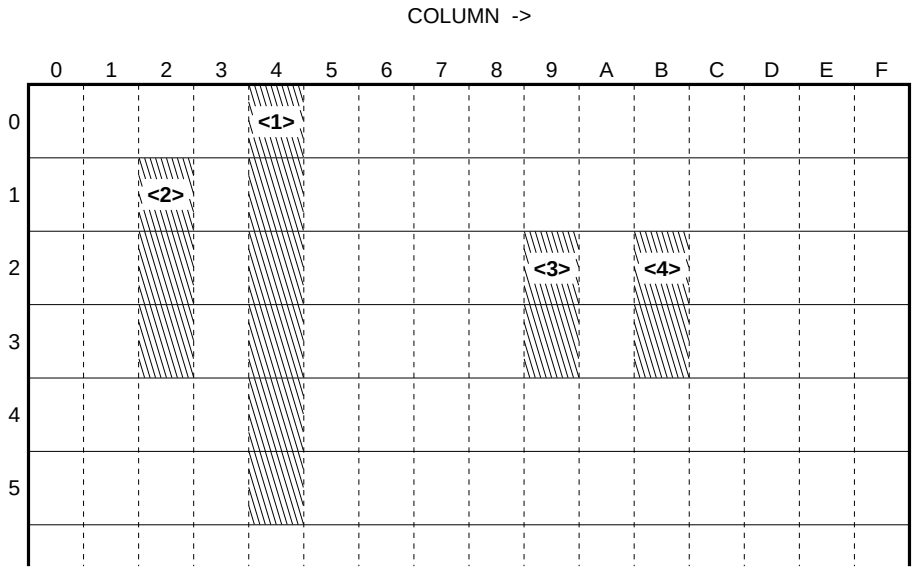
- Nibble information: Indicates the storage area (number of nibbles) for the symbol.
  For the horizontal type, the value obtained by subtracting 1H from the actual number of nibbles is stored. For the vertical type, the value obtained by adding 7H to the number of nibbles is stored.
  When a symbol is defined, the value of the nibble information is automatically stored by the assembler.

- Bank : Bank number of the data memory (RAM)
  The above is stored by a symbol definition pseudo instruction.
- Address : Address in data memory (RAM)
  The above is stored by a symbol definition pseudo instruction.
- Example of a data memory type symbol



<1> 0.02H-0.03H .... NIBBLE2 (define 0.02H.)
<2> 0.0EH-0.11H .... NIBBLE4 (define 0.0EH.)
<3> 0.16H-0.1AH .... NIBBLE5 (define 0.16H.)

With RA17K, a symbol can be defined for a vertical data memory area. In this case, the pseudo instruction is NIBBLE2V to NIBBLE8V. V indicates Vertical (vertical type).



<1> 0.04H-0.54H .... NIBBLE6V (define 0.04H.)
<2> 0.12H-0.32H .... NIBBLE3V (define 0.12H.)
<3> 0.29H-0.39H .... NIBBLE2V (define 0.29H.)
<4> 0.2BH-0.3BH .... NIBBLE2V (define 0.2BH.)

The evaluation value of a symbol defined by the vertical type nibble pseudo instruction (NIBBLEnV) has a value obtained by adding 7H to the number of nibbles as nibble information of the high-order four bits.

This is because vertical data memory is differentiated from horizontal data memory.

```
NIBBLE2V   ...   9H
NIBBLE3V   ...  0AH
    ⋮
NIBBLE8V   ...  0FH
```

The data memory type symbol definition is explained.

[Format]

All formats are the same. Use a type name to which a pseudo instruction is added.

```
<name>ΔMEMΔ<expression>[Δ][;<comment>]
<name>ΔNIBBLEnΔ<expression>[Δ][;<comment>]
<name>ΔNIBBLEnVΔ<expression>[Δ][;<comment>]
```

[Function]

Assigns the value of <expression>, described in the operand, to <name>.
<name> is used as a data memory type symbol.

[Explanation]

(1) The following can be described in <expression>:

<1> <expression> of data memory type

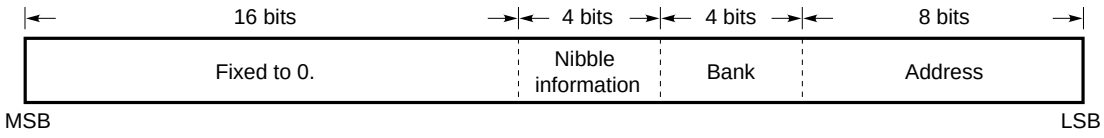
<2> Binomial expression containing one bit-delimiting operator

```
<bank-expression>.<address-expression>
```

Both the <bank-expression> and <address-expression> are data type expressions. The evaluation value of <bank-expression> in the first item specifies the bank in data memory. <address-expression> in the second item specifies the address.

When describing a numeric expression, describe both the bank of the address to be specified and the address itself. A period (.) must be inserted between the bank and the address. This period is called a bit-delimiting operator. A blank can be inserted both before and after the bit-delimiting operator.

(2) The evaluation value of <expression> consists of 32 bits. It contains the nibble information, bank, and address value. If the evaluation value of <bank> produces a bank number that is not installed in the target device, an error (F046: Invalid BANK No.) occurs. If the evaluation value of <address> produces an area that is not installed in the target device (for a multi-nibble, all nibbles must be contained in the installed area), an error (F067: Address error) occurs and the evaluation value is registered as 0.



(3) When a symbol name is described in the operand, the number of nibbles in the symbol can differ from that in a symbol definition pseudo instruction. In other words, the following description is valid:

```
SYM1    NIBBLE4    0.05H
SYM2    NIBBLE8    SYM1
```

In this case, the evaluation values of symbols SYM1 and SYM2 are as follows:

```
SYM1    :   (00003005H)
SYM2    :   (00007005H)
```

(4) A blank must be inserted between <name> and the pseudo instruction, and also between the pseudo instruction and <expression>. If no blank is inserted, this line is not interpreted as a MEM pseudo instruction and an error occurs (F037: Syntax error). This line is invalidated and the symbol described in <name> is not registered.

(5) If the pseudo instruction description is invalid, an error (F037: Syntax error) occurs. This line is invalidated and the symbol described in <name> is not registered.

(6) All operators can be used in <expression>. Note, however, that the bit delimiting operator (.) is used to separate the bank and address.

(7) If an <expression> other than the data memory type is described, an error (F045: Invalid type) occurs and <expression> is registered as evaluation value 0.

(8) A data memory type symbol can be used as an operand of an instruction in which the data memory type must be specified.
If a multi-nibble symbol is described in an operand, nibble information and bank information is not reflected in the instruction. In other words, only an address value indicated by the low-order eight bits is reflected in an object code.

(9) Only the low-order 16 bits are valid as the evaluation value of <expression>, described in an operand. The high-order 16 bits are unconditionally cleared to zero.

(10) Even if nibble information is contained in the evaluation value of <expression> of an operand in the MEM pseudo instruction, its value is not reflected.
In other words, 0 is always set in the nibble information area.

8.6 FLG PSEUDO INSTRUCTION

[Format]

```
<name>ΔFLGΔ<expression>[Δ][;<comment>]
```

[Function]

Assigns the value of <expression>, described in an operand, to <name>.
<name> is used as a flag type symbol.

[Explanation]

(1) The following can be described in <expression>:

<1> Flag type <expression>

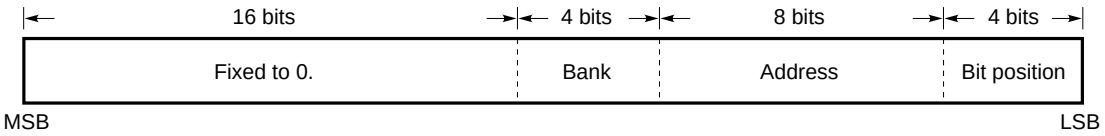
<2> Binominal expression containing two bit-delimiting operators

```
<bank-expression>.<address-expression>.<bit-expression>
```

<bank-expression>, <address-expression>, and <bit-expression> are data type expressions. The evaluation value of <bank-expression> in the first item specifies the bank. <address-expression> in the second item specifies an address. <bit-expression> in the third item specifies a bit position.

To describe a numeric expression, describe the bank, address, and bit position. The bank, address, and bit position must be separated by a period (.). This period is called a bit-delimiting operator. A blank (space or TAB) can be inserted before and after the bit-delimiting operator.
The numerics that can be described in <bit-position> are 0, 1, 2, and 3 only.

(2) The evaluation value of <expression> consists of 32 bits. It contains the bank, address, and bit position values. If the evaluation value of <bank> produces a bank that is not installed in the target device, an error (F046: Invalid BANK No.) occurs. If the evaluation value of <address> produces an area that is not installed in the target device (for a multi-nibble, all nibbles must be contained in the installed area), an error (F067: Address error) occurs and the evaluation value is registered as 0.
The evaluation value of a bit position (0, 1, 2, 3) is the value of the bit (1, 2, 4, 8).
If an evaluation value of <bit-position>, that is, the value of the low-order four bits, is other than 1, 2, 4, or 8H, an error (F044: Invalid value) occurs and the evaluation value is registered as 0.



(3) A blank must be inserted between <name> and the pseudo instruction, and also between the pseudo instruction and <expression>. If no blank is inserted, this line is not recognized as a flag type symbol definition pseudo instruction and an error (F037: Syntax error) occurs. This line is invalidated and the symbol described in <name> is not registered.

(4) If the pseudo instruction description is invalid, an error (F037: Syntax error) occurs. This line is invalidated and the symbol described in <name> is not registered.

(5) All operators can be used in <expression>. Note, however, that the bit delimiting operator (.) is used to separate the bank, address, and bit position.

(6) If the evaluation value of <expression> is other than the flag type, an error (F045: Invalid type) occurs. The evaluation value is registered as 0.

(7) Only the low-order 16 bits are valid as the evaluation value of <expression>, described in the operand. The high-order 16 bits are unconditionally cleared to zero.

8.7 SET PSEUDO INSTRUCTION

[Format]

<name>ΔSETΔ<expression>[Δ][;<comment>]

[Function]

Assigns the value and type of <expression>, described in the operand, to <name>.
The value registered in <name> can be changed using the SET pseudo instruction.
The same type as that of <expression> in the operand is defined in <name>. If <expression> of the operand is a numeric constant or numeric expression, <name> is used as the data type symbol.

[Explanation]

(1) A blank (space or TAB) must be inserted between <name> and the pseudo instruction, and also between the pseudo instruction and <expression>. If no blank is inserted, this line is not recognized as a SET pseudo instruction and an error (F037: Syntax error) occurs. This line is invalidated and the symbol described in <name> is not registered.

(2) If the pseudo instruction description is invalid, an error (F037: Syntax error) occurs. This line is invalidated and the symbol described in <name> is not registered.

(3) The type of <name>, defined by the SET pseudo instruction, cannot be changed within the same module to another type by using the SET pseudo instruction. If an attempt is made to make this change, an error (F045: Invalid type) occurs and this line is invalidated. The symbol specified in <name> continues to hold the value used when it was previously defined.

(4) A symbol defined by the SET pseudo instruction cannot be referenced by an external module. In other words, if the symbol is subjected to PUBLIC declaration, an error (F167: Invalid PUBLIC statement) occurs. However, a symbol other than those defined by the SET pseudo instruction is subjected to PUBLIC declaration.

[Example]

```
PUBLIC  A, B, C, D, E
C    SET    3
```

In this case, an error occurs in the PUBLIC declaration for the C symbol. However, symbols A, B, D, and E, but not C, are subjected to PUBLIC declaration.

(5) The symbol defined by the SET pseudo instruction is not output to the memory map and flag map. The value defined at the end of the program is output to the cross reference.

[MEMO]

CHAPTER 9 DATA DEFINITION PSEUDO INSTRUCTIONS

This chapter explains the data definition pseudo instructions used to generate the data to be described in program memory. The data definition pseudo instructions are DB, DW, and DCP. DB and DW convert an expression described in an operand to data, in units of bytes or words. DCP generates data used for the peripheral circuit IDC of a device. The coded data is output to the object file as is.

Data definition pseudo instructions are:

- DW pseudo instruction (define data in words.)
- DB pseudo instruction (define data in bytes.)
- DCP pseudo instruction (define IDC data.)

[Notes]

(1) In relocatable mode, a data definition pseudo instruction can be described only within the section and table blocks. (See **Chapter 10**.)

(2) If the <label> description is invalid, an error (see **Section 4.3**) occurs. One line is invalidated and an object code for the NOP instruction is created.

9.1 DW (DEFINE WORD) PSEUDO INSTRUCTION

[Format]

```
[<label>:][Δ]DWΔ<expression>[Δ][;<comment>]
```

[Function]

Stores the evaluation value of <expression>, described in the operand, into the address indicated by the current location counter. <expression> can be used to describe a numeric expression, data type symbol, or character constant consisting of one shift JIS character or one half-size character.

[Notes]

(1) The operand can describe only one <expression> or <symbol> that can be expressed by 16 bits. If the evaluation value of <expression> exceeds 16 bits, a warning (W169: Omitted a surplus due to an input value is over a regular value) occurs and the 17th and subsequent bits are truncated. If two or more operands are specified with each separated by a comma (,), an error (F037: Syntax error) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(2) Only one full-size or half-size character can be described as a character constant enclosed by single quotation marks ('). If two or more characters are enclosed in single quotation marks as a character constant, an error (F051: Invalid data length) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(3) If a symbol described in an operand is not defined, an error (F058: Undefined symbol) occurs. This line is invalidated and the NOP instruction is generated. An external definition symbol and forward reference symbol can be described.

(4) Only a data type expression can be described in <expression> of the operand. If an expression other than the data type expression is described, an error (F045: Invalid type) occurs. This line is invalidated and an object code of the NOP instruction is generated.
To describe a symbol other than a data type symbol, the symbol type must be converted to a data type by using the type conversion function.

(5) If an expression described in <expression> is incorrect, an error (see **Section 4.7.1** for details) occurs, and an object code for the NOP instruction is generated.

(6) If the evaluation value of an expression described in an operand consists of fewer than 16 bits, it is stored starting from the least significant bit. In other words, 0 is stored in the high-order bits and the evaluation value is stored in the low-order bits.

(7) If <expression> is omitted or two or more <expression> items are described, an error (F037: Syntax error) occurs. This line is invalidated and an object code of the NOP instruction is generated.

[Example]

```
E      LOC.    OBJ.   M  I  CL    SOUCE STATEMENT

       00000   1923              DW     1800H  OR  123H
       00001   0041              DW     'A'
       00002   889F              DW     '亜'
       00003   000D              DW     1101B
       00004   0000              DW     0001H  +  0FFFFH

F051   00005   3CF0              DW     'AB'
F037   00006   3CF0              DW     'A','B'
       00007   ****              DW     UNDEF
```

9.2 DB (DEFINE BYTE) PSEUDO INSTRUCTION

[Format]

```
[<label>:][Δ]DBΔ<expression>[,<expression>[...]][Δ][;<comment>]
```

[Function]

Converts <expression>, described in the operand, to 8-bit (1-byte) data. <expression> can describe a numeric expression, data type symbol, and character constant consisting of a shift JIS kanji character or half-size character.

[Notes]

(1) Any number of operands can be described, each separated by a comma (,), provided the maximum number of characters (255) is not exceeded.

(2) If a symbol described in an operand is undefined, an error (F058: Undefined symbol) occurs. This line is invalidated and an object code of the NOP instruction is generated.
However, external definition and forward reference symbols can be described.

(3) If an expression described in <expression> is incorrect, an error (see **Section 4.7.1** for details) occurs. An object code of the NOP instruction is generated.

(4) If the evaluation value of an expression described in an operand consists of 8 bits, 00H is stored in the low-order eight bits and the evaluation value is stored in the high-order eight bits.

(5) If the value of <expression> consists of nine or more bits, a warning (W169: Omitted a surplus due to an input value is over a regular value) is output and only the data for the low-order eight bits is set.

(6) If <expression> is omitted, an error (F037: Syntax error) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(7) When the generation of two or more object codes is described, if an error (such as that for an undefined symbol) occurs in the portion in which a forward reference symbol is described, an object code for the NOP instruction is generated in this portion only.

(8) Only a data type expression can be described in <expression> of an operand. If an expression other than the data type expression is described, an error (F045: Invalid type) occurs. This line is invalidated and an object code of the NOP instruction is generated.

To describe a symbol other than a data type symbol, the symbol type must be converted to a data type.

[Example]

```
E       LOC.    OBJ.   M  I  STATEMENT

        00000   2300        DB      1800H  OR  123H   Warning
        00001   0000        DB      1800H  AND 0FF00H Warning
        00002   4100        DB      'A'
        00003   4142        DB      'A','B','C'
        00004   4300
        00005   4142        DB      'ABC'
        00006   4300
        00007   889F        DB      '亜唖娃阿'
        00008   88A0
        00009   88A1
        0000A   88A2
        0000B   0102        DB      1,2,3,4,5,6,···
        0000C   0304
        0000D   0506
                 ⋮
        00010   FF00        DB      1800H  +  0FFFFH  Warning
        00013   ****        DB      UNDEF
```

9.3 DCP (DEFINE CHARACTER PATTERN) PSEUDO INSTRUCTION

[Format]

```
[<label>:][Δ]DCPΔ<expression>,'<pattern>'[Δ][;<comment>]
```

[Function]

Acquires the pattern described in the second operand as 10-bit object data. This data is used as the pattern for display data (one-character width, one horizontal column) of the image display controller (IDC). Therefore, the DCP pseudo instruction need not be used for a device that does not support the IDC function. See the data sheet for the device for details of the IDC function.
The definition is made as follows:

(1) <expression> can describe a numeric expression and data type symbol having a value of 0H or 1H. <expression> specifies whether outline processing is performed for the defined pattern in the second operand.

<expression> = 0H: Does not perform outline processing.
<expression> = 1H: Performs outline processing.

If the evaluation value of <expression> is other than 0H or 1H, an error (F044: Invalid value) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(2) <pattern> can describe only three half-size characters "O" (alphabetic character), "#", and " " (space). <pattern> must consist of 10 characters, using only these half-size characters.
The meanings of the three half-size characters are as follows:

O : Lighting data of one dot
: Outline of one dot
" " : Non-lighting data of one dot

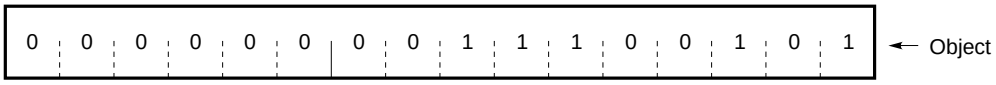
If <pattern> does not consist of 10 characters, an error (F083: Illegal format) occurs. This line is invalidated and the NOP instruction is generated. If a character other than the above half-size characters is described, an error (F082: Illegal character) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(3) If the evaluation value of the first operand is 0, an object is generated according to the following rules:

<1> 0 is unconditionally inserted into the six high-order bits of the object.
<2> 1 is inserted into the 10 low-order bits at the location corresponding to the bits in which O of the second operand is described.

[Example]

```
DCP   0,' OOO   O O'
```



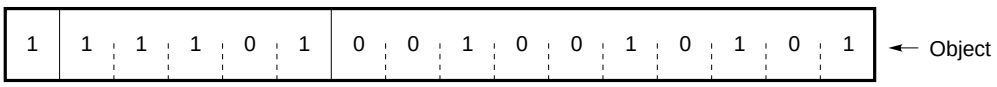
If the evaluation value of the first operand is 0, <pattern> cannot describe #. If # is described, an error (F082: Illegal character) occurs. This line is invalidated and the object code of the NOP instruction is generated.

(4) If the evaluation value of the first operand is 1, the object is generated according to the following rules:

<1> 1 is unconditionally inserted into the high-order position of the object.
<2> 1 is inserted in the low-order 10 bits (bit#9-bit#0), corresponding one-to-one to # (outline symbol).
<3> Bits#14-bit#10 have a one-to-one correspondence with a block in which the second operand characters are separated two at a time. If at least one O appears in this block, 1 is inserted into the bits of the associated object. However, O and " " cannot coexist in one block. If O and " " coexist, an error (F083: Illegal format) occurs. This line is invalidated and an object code of the NOP instruction is generated.

[Example 1]

```
DCP 1,' 00#00# #0#'
```



[Example 2] For an F error

```
DCP   1,'OOO   O O#'
```

-> O and " " coexist in one block.

(5) The DCP pseudo instruction cannot be coded in an address whose low-order four bits are 0FH. The assembler cannot determine the address where an instruction is placed. This check is made by the linker (LK17K).

(6) If the coded <expression> is invalid, an error (see **Section 4.7.1** for details) occurs. An object code of the NOP instruction is generated.

[Notes]

(1) If an undefined symbol is described in <expression>, an error (F058: Undefined symbol) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(2) If the EXTRN symbol is described in <expression>, an error (F150: Impossible to write the external symbol) occurs. This line is invalidated and an object code of the NOP instruction is generated. This is because if <expression> is not evaluated, the second operand cannot be checked.

(3) If one or fewer operands, or three or more are specified, an error (F037: Syntax error) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(4) When <expression> is evaluated, if the type is not a data type, an error (F045: Invalid type) occurs. This line is invalidated and an object code of the NOP instruction is generated.

(5) If <pattern> is other than a character constant, an error (F037: Syntax error) occurs. This line is invalidated and an object code of the NOP instruction is generated.

CHAPTER 10 PROGRAM CONFIGURATION PSEUDO INSTRUCTIONS

This chapter explains the pseudo instructions used to define program configurations. The pseudo instructions used in relocatable mode differ from those used in absolute mode. The absolute mode basically conforms to conventional AS17K. In relocatable mode, a section block can be defined with the CSEG pseudo instruction. Section blocks are used as the minimum unit of relocation during linking. The allocation method can be indicated in a program by using the pseudo instructions.

The program configuration pseudo instructions are as follows:

- CSEG pseudo instruction (used to define a segment or section block)
  (1) A segment is defined in absolute mode.
  (2) A section block is defined in relocatable mode.

- END pseudo instruction (used to indicate the end of a program)
- ENSURE pseudo instruction (used to set bank, register, and pointer values. Usable only in relocatable mode.)

10.1 CSEG PSEUDO INSTRUCTION (ABSOLUTE MODE)

[Format]

CSEG∆<expression>[∆][;<comment>]

(<expression>=0-7: depends on the target device)

[Function]

Declares the start of the segment specified in <expression>. In other words, the programs appearing between one CSEG pseudo instruction and the next are handled as being within the same segment.

[Notes]

(1) The above format can be described only in absolute mode.
If this format is described in relocatable mode, an error (F112: Impossible to use on relocatable mode) occurs. This line is invalidated.

(2) To describe two or more CSEG pseudo instructions within a source program, specify the <expression> values in ascending order, regardless of whether the source program is split.
If <expression> is equal to or less than the segment number which belongs to the instruction and is that which was described most recently, an error (F044: Invalid value) occurs. The CSEG pseudo instruction is ignored and is assumed to be the same segment as that described most recently. Consecutive addresses are generated (in this case "consecutive" means that a patch area is used if the size of the program exceeds that of the user area of the segment).

(3) If the evaluation value of <expression> corresponds to the number of a segment that is not installed in a target device, an error (F044: Invalid value) occurs and the line is invalidated. Therefore, the instruction is assumed to correspond to the same segment as that described most recently and, therefore, consecutive addresses are generated.

(4) A warning (see **Section 2.3**) is output if an instruction for generating an object preceding the CSEG pseudo instruction is not a branch instruction (BR), return instruction (RET, RETI, RETSK), or data definition instruction (DW, DB, DCP).

(5) When the CSEG pseudo instruction is described, the first address of the segment is set in the location counter. The use of this counter starts from the line subsequent to the CSEG pseudo instruction.

(6) If an undefined symbol or forward reference symbol is described in <expression>, an error (F058: Undefined symbol) occurs. Consecutive addresses are generated immediately after the preceding segment.

(7) If the EXTRN symbol is described in <expression>, an error (F150: Impossible to write the external symbol) occurs. This line is invalidated and consecutive addresses are generated immediately after the preceding segment.

(8) If the <expression> description is incorrect, an error (see **Section 4.7.1** for details) occurs. This line is invalidated and consecutive addresses are generated immediately after the preceding segment.

(9) If the operand is omitted or two or more operands are described, an error (F037: Syntax error) occurs. This line is invalidated and consecutive addresses are generated immediately after the preceding segment.

(10) If a label is described, an error (F037: Syntax error) occurs. This line is invalidated and consecutive addresses are generated immediately after the preceding segment.

(11) Only a data type expression can be described in <expression> of the operand. If an expression other than the data type is described, an error (F045: Invalid type) occurs. This line is invalidated and consecutive addresses are generated immediately after the preceding segment.

[Relationship between CSEG and linking]

(1) The CSEG pseudo instruction can be omitted. If so, the instruction is linked with the last segment of the preceding object file (indicated in the order of the files specified at linking).

```
Source 1              Source 2
CSEG         0        ADD M1, #1
                      END
NOP
NOP
END
```

If source 1 then source 2 are linked, the object of source 2 is linked with the same segment (segment 0) as that of the preceding file (source 1).

(2) Based on the same idea as that in (1), at linking, the instruction that generater an object code that exists until the CSEG pseudo instruction is described is linked with the same segment as that of the preceding file.

```
Source 1              Source 2
CSEG         0        ADD M1, #1
NOP                   CSEG     2
NOP                   INC IX
END                   END
```

If source 1 then source 2 are linked, the ADD instruction of source 2 is linked with the same segment (segment 0) as that of the preceding file and the INC instruction is allocate in segment 2.

(3) If a file from which the CSEG pseudo instruction has been omitted is specified first at linking, segment 0 is automatically set.

10.2 CSEG PSEUDO INSTRUCTION (RELOCATABLE MODE)

[Format]

<1> <section-name> Δ CSEG[ΔATΔ<expression>]

<section-name> - Section-name
<expression> - Expression of the absolute address

<2> <section-name> Δ CSEG[Δ<allocation-specification>][Δ<expression>]

<section-name> - Section name
<allocation-specification> - SSYS : System call subroutine (indirect allocation**Note 1**)
DSYS : System call subroutine (direct allocation**Note 2**)
SBR : Subroutine (indirect allocation**Note 1**)
DSBR : Subroutine (direct allocation**Note 2**)
CROM : Non-program memory data (data to be stored in ROM other than program memory)
TABLE : Data for table lookup (data to be referenced with the MOVT instruction)
<expression> - Allocation segment number
0-7 or SYS

<3> <section-name> Δ CSEG[Δ<allocation-specification>]

<section-name> - Section name
<allocation-specification> - BOOT : Main routine (boot section)
VECTn : Interrupt processing routine (indirect allocation**Note 1**)
DVECTn : Interrupt processing routine (direct allocation**Note 2**)
However, n is hexadecimal 1-9, 0A-0F, or 10-3F.

Notes 1. A branch instruction is generated in the area in which a section, for which allocation has been specified, is to be allocated. The section is allocated within an area that can be referenced by the branch instruction.

2. The section for which allocation was specified is directly allocated within the area in which the section is to be allocated.

[Function]

Defines the minimum unit of the reallocation at linking.

An instruction, described between one CSEG and the next, between a CSEG and END, or between a CSEG and the end of the file, is used as a single section block. It is used as the minimum unit for reallocation at linking.

An instruction which generates an object code must be described in this section block.

The linker performs allocation processing for each allocation specification. Refer to the **Linker (LK17K) User's Manual** for details of allocation processing.

[Explanation]

(1) An instruction which generates an object code must be described within the section block defined between one CSEG pseudo instruction and the next, between a CSEG instruction and an END pseudo instruction, or between a CSEG instruction and the end of the file. If the instruction is described outside the section block, an error (F146: Impossible to write out of section block) occurs and no object code is generated.

(2) If a reserved symbol is described in <section-name>, an error (F037: Syntax error) occurs. This line is invalidated.

(3) If a defined symbol is described in <section-name>, an error (F057: Symbol multi defined) occurs. This line is invalidated.

(4) If <section-name> is omitted, an error (F037: Syntax error) occurs. This line is invalidated.

[Notes]

(1) The ORG pseudo instruction, defined in the section, is specified in an offset address for which the beginning of the section block is address 0H.

(2) A name described in <section-name> is registered as a symbol. Unlike general symbols, this symbol cannot be described in an operand of other instructions and other pseudo instructions. If described, an error (F182: Specified illegal symbol name) occurs. If this symbol is described as an operand of an instruction which generates an object code, the NOP instruction is generated. If it is described as an operand of an pseudo instruction, the pseudo instruction is invalidated.

(3) Section blocks having the same name cannot be defined in a single source module file. If two or more section blocks having the same name are defined, an error (F197: Section or table block multi defined in a source file) occurs in the second CSEG and subsequent statements. This line is invalidated.

(4) If an undefined symbol or forward reference symbol is described in <expression>, an error (F058: Undefined symbol) occurs. This line is invalidated.

(5) If the EXTRN symbol is described in <expression>, an error (F150: Impossible to write the external symbol) occurs. This line is invalidated.

(6) If the <expression> description is invalid, an error (see **Section 4.7.1** for details) occurs. This line is invalidated.

(7) A data type expression is described in <expression> of the operand in format **<1>**. If other expressions are specified, an error (F044: Invalid value) occurs. This line is invalidated.

(8) A data type expression or "SYS" is described in <expression> of the operand in format **<2>**. If other expressions are specified, an error (F044: Invalid value) occurs. This line is invalidated.

(9) If an evaluation value of <expression> in format **<2>** is the number of a segment that is not installed in a target device, an error (F044: Invalid value) occurs. This line is invalidated.

(10) Only hexadecimal values between 1H and 3FH can be specified as n of VECTn and DVECTn in <allocation-specification> of format **<3>**. However, H, indicating the end of a hexadecimal number, is not described.

(11) If the operands are omitted, the linker automatically performs allocation in any segment.

[Examples]

[Example 1] Section definition

- **Definition with the operands omitted**

```
R01    CSEG
        ⋮
       END
```

Section R01 is allocated in any segment at linking because no segment has been specified.

- **Defining only the allocation specification in format <2>**

```
R02    CSEG     SBR
        ⋮
       END
```

Section R02 has a subroutine attribute. It is allocated in any segment at linking because no segment has been specified.

- **Definition with the allocation specification omitted from format <2>**

```
R02    CSEG     0
        ⋮
       END
```

Section R02 is allocated in segment 0.

- **Defining the boot section**

```
B01    CSEG     BOOT
        ⋮
       END
```

Section B01 has a boot attribute. At linking, it is allocated as the section to be executed first after resetting.

- **Defining the interrupt handling routine**

```
I01    CSEG     VECT1
        ⋮
       END

I13    CSEG     VECT13
        ⋮
       END

I32    CSEG     VECT32
        ⋮
       END
```

At linking, Section I01 is allocated as the interrupt handling routine for interrupt vector address 1H.
At linking, section I13 is allocated as the interrupt handling routine for interrupt vector address 13H.
At linking, section I32 is allocated as the interrupt handling routine for interrupt vector address 32H.

10.3 END PSEUDO INSTRUCTION

[Format]

```
END
```

[Function]

Indicates the end of the program to the assembler. The description, however, can be omitted.

[Notes]

(1) No more than one END statement may be used for a single assemble unit. An error does not occur even if the last line of the source module file does not end with an END statement or if the END statement is omitted.

(2) In a source module file, the END statement must be described at the end. If a statement follows END, a warning (W065: Statement after END) is output for that line. If this warning is output, the statements after END are not analyzed. Therefore, only the END description line and the next line, for displaying W065, are output to the intermediate list. No data is subsequently output.

(3) If the END statement is omitted, a warning (W028: No END statement) is output.

(4) The END statement is processed by the conditional assembly pseudo instruction, even when assembly is skipped. (See **Chapter 13**.)

(5) If the END statement is described in the macro definition (macro body), assembly terminates with the END statement at macro registration. Therefore, any statements after the END statement are not interpreted and are not output to the intermediate list. See (2) above for details of output to the intermediate list.

(6) If a label or operand is described in the END statement, an error (F037: Syntax error) occurs. This line is invalidated and thus not interpreted as an END statement.

(7) If the END statement is described in the include file, the include file is included normally. If an instruction that generates an object code is placed after the END statement, the object code is generated normally. However, when control returns to the source module file, if any statement follows an INCLUDE statement in which a file containing an END statement is described, a warning (W065: Statement after END) is output.

If a warning is output

Source module file A.ASM

```
      :
INCLUDE 'B.ASM'
NOP <- Warning output line
      :
```

Include file B.ASM

```
      :
END
```

The warning is output because the statement is described after INCLUDE 'B.ASM'.

If no warning is output

Source module file A.ASM

```
      :
INCLUDE 'B.ASM'
(End of the file)
```

Include file B.ASM

```
      :
END
NOP
NOP
```

No warning is output, even if a statement is described after the END statement in an include file. For a source module file, no warning is output because no statements are described after INCLUDE in the file in which the END statement is described.

(8) If the END statement is described in the summary statement, the assembler terminates at this line. A warning (W162: No ENDSUM statement) is output.

10.4 ENSURE PSEUDO INSTRUCTION

[Format]

```
ENSURE△register-name=<expression>[,register-name=<expression>]···
```

register-name = {BANK|RP|AR}
<expression> = {BANK value|RP value|AR value|NOCHANGE|UNKNOWN}

[Function]

Issued for the assembler and specifies bank, register pointer (RP), and address register (AR) values immediately after a label line and CALL instruction.

<expression> :	Each register value	... Described only when it belongs to the related register.
	NOCHANGE	... Described when a value is unclear but the previous value remains unchanged.
	UNKNOWN	... Described when a value is unclear and it is impossible to determine whether the previous value has changed.

[Explanation]

(1) When a bank value is set by ENSURE, it is also set in ZZZBANK.

(2) When an RP value is set by ENSURE, it is also set in ZZZRP.

(3) When an AR value is set by ENSURE, it is also set in ZZZAR.

(4) When NOCHANGE is set in <expression>, it is determined that a register value specified for register-name remains as is and that an assemble time variable is not changed.

(5) When UNKNOWN is specified in <expression>, the register value specified for register-name may have been changed. –1 is set as the assemble time variable.

(6) When ENSURE is described, describe only those registers whose contents are known. Any other registers should not be described. For any registers that are not described, –1 is set for the corresponding assembly time variables.

(7) ENSURE can be described only in relocatable mode. If an attempt is made to use ENSURE in absolute mode, an error (F113: Impossible to use on absolute mode) occurs. This line is invalidated.

[Notes]

(1) If an undefined symbol is described in <expression>, an error (F058: Undefined symbol) occurs. This line is invalidated. ZZZBANK, ZZZRP, and ZZZAR all continue to hold their previous values.

(2) If the EXTRN symbol was described in <expression>, an error (F150: Impossible to write the external symbol) occurs. This line is invalidated. ZZZBANK, ZZZRP, and ZZZAR all continue to hold their previous values.

(3) If the <expression> description is invalid, an error (see **Section 4.7.1** for details) occurs. This line is invalidated. ZZZBANK, ZZZRP, and ZZZAR all continue to hold their previous values.

(4) If a bank or register pointer that is not installed is specified, an error (F044: Invalid value) occurs. This line is invalidated.

(5) If a label is specified in ENSURE, an error (F037: Syntax error) occurs.

(6) If a character other than BANK, RP, or AR is specified as a register name, an error (F050: Not reserved word) occurs.

(7) To specify a value using <expression>, use a data type expression. If an expression other than the data type is described, an error (F045: Invalid type) occurs. This line is invalidated.

[MEMO]

CHAPTER 11 LOCATION COUNTER CONTROL PSEUDO INSTRUCTION

11.1 ORG

[Format]

```
[<label>:][Δ]ORGΔ<expression>
```

[Coding example]

In relocatable mode

When using ORG, specify an offset for which the beginning of the section block is address 0H. Therefore, if ORG 100H is specified, the location address of the next line is 100H.

```
LOC.    SOURCE  STATEMENT

        SEC1    CSEG
                 :
  21             AND     R1,M1
  22             ORG     100H
 100             AND     R1,M1
                 :
                END
```

[Function]

Sets an evaluation value for <expression> in the location counter.

A location counter value set by the ORG pseudo instruction is used from the line subsequent to that on which the ORG pseudo instruction is described. Therefore, the label value described in the ORG pseudo instruction is not that value specified by <expression>, instead being the preceding value (on the line on which the ORG pseudo instruction was described).

[Notes]

(1) In relocatable mode, the ORG pseudo instruction can be described only in the section block. If it is described outside the section block, an error (F146: Impossible to write out of section block) occurs. This line is invalidated.

(2) In relocatable mode, the ORG pseudo instruction defined within the section is positioned to the offset address for which the beginning of the section block is 0H.

(3) If the <expression> description is invalid, an error (see **Section 4.7.1** for details) occurs. This line is invalidated. The location counter value remains as is.

(4) If an undefined symbol or forward reference symbol is described in <expression>, an error (F058: Undefined symbol) occurs. This line is invalidated. Therefore, the location counter value remains as is.

(5) If an external definition symbol is described in <expression>, an error (F150: Impossible to write the external symbol) occurs. This line is invalidated. Therefore, the location counter value remains as is.

(6) If a value less than the present location value is specified in <expression>, an error (F048: ORG address error) occurs. This line is invalidated. Therefore, the location counter value remains as is.

(7) In absolute mode, if an address (outside the ROM and EPA areas) that does not exist in a target device is specified, an error (F152: The address is out of ROM) occurs. The location counter value remains as is.

(8) In absolute mode, specify an absolute address corresponding to the segment specified by the CSEG pseudo instruction.

(9) If the operand is omitted or two or more operands are specified, an error (F037: Syntax error) occurs.

(10) If the <label> description is invalid, an error (see **Section 4.3**) occurs.

(11) Only label type <expression> or data type <expression> can be described in <expression>. If an expression other than the label type or data type expressions is specified, an error (F045: Invalid type) occurs. This line is invalidated. The location counter value remains as is.

CHAPTER 12 EXTERNAL DEFINITION AND EXTERNAL REFERENCE PSEUDO INSTRUCTIONS

12.1 PUBLIC, PUBLIC BELOW ... ENDP

[Format]

(1) [<label>:][Δ]PUBLICΔ<symbol>[,<symbol>...][Δ][;<comment>]

(2) [<label>:][Δ]PUBLICΔBELOW[Δ][;<comment>]

```
             Name  Symbol definition pseudo instruction Expression
               ⋮                        ⋮                    ⋮
                                      ENDP                      [;<comment>]
```

[Coding examples]

```
 Symbol    Mnemonic        Operand
 DATA1     DAT             2H
 FLAG2     FLG             0.10H.3
           ⋮
           PUBLIC          DATA1,FLAG2
           ⋮
           PUBLIC          BELOW
 MEM00     MEM             0.00H
 MEM10     MEM             0.10H
           ⋮
           ENDP
```

```
 Symbol    Mnemonic        Operand
           PUBLIC BELOW
 DATA1     DAT             2H
 FLAG2     FLG             0.10H.3
           ⋮
           PUBLIC          DATA1,FLAG2 <- No error occurs.
           ⋮
           PUBLIC BELOW                <- No error occurs because nesting is not
                                          supported.
 MEM00     MEM             0.00H
 MEM10     MEM             0.10H
           ⋮
           ENDP
```

[Function]

Declares that a symbol described in the operand field of the PUBLIC declaration statement or a symbol defined in a block enclosed by PUBLIC BELOW and ENDP is to be referenced by other modules (external modules) created when a program was split as part of modularization.

[Notes]

(1) The PUBLIC statement can be described in any location in a source program (or within a macro).

(2) A symbol declared by PUBLIC must be defined as a symbol within the same assembly unit (within a source module in which the declaration is made). If not defined, an error (F058: Undefined symbol) occurs. If, however, two or more symbols are described in a single line, a symbol other than the erroneous symbols is registered (PUBLIC declaration).

(3) Any number of symbols can be described in the operand field of the PUBLIC statement, up to the maximum number of characters that can be specified on one line (255 characters). If a line is fed, those symbols on the second and subsequent lines are not registered. PUBLIC must be declared again.

(4) Two or more symbols having different attributes are described in the operand field of the PUBLIC statement.

(5) An instruction other than a symbol definition pseudo instruction can also be described in a block enclosed by PUBLIC BELOW and ENDP.

(6) If there is no corresponding ENDP for PUBLIC BELOW, an error (F033: No ENDP statement) occurs in the END statement or at the end of a file.

(7) If ENDP is described without PUBLIC BELOW, an error (F085: Invalid ENDP statement) occurs. This line is invalidated.

(8) The symbol types that can be specified in <symbol> are the data type, label type (including a label defined by symbol:), data memory type, and flag type only. If other symbol types are declared by PUBLIC, an error occurs. However, the symbols specified by the SET pseudo instruction between PUBLIC BELOW and ENDP are excluded from the PUBLIC declaration.
A list of symbols that cannot be declared, together with the related error message, are shown below.

Symbols that cannot be declared**Note**

- Symbols defined by the SET pseudo instruction
- Reserved words
- Local symbols within macros

Error messages (F167: Invalid PUBLIC statement)

Note A section name (see **Section 10.2**) cannot be declared by PUBLIC. If declared, an error (F182: Specified illegal symbol name) occurs.

(9) An error does not occur even if the same symbol is declared two or more times with PUBLIC or PUBLIC BELOW.

(10) If a label or operand is described in the ENDP statement, an error (F037: Syntax error) occurs. This line is invalidated.

(11) If an operand is omitted from the PUBLIC statement, an error (F037: Syntax error) occurs.

(12) If an operand is described in the PUBLIC BELOW statement, an error (F037: Syntax error) occurs.

(13) If the <label> description is invalid, an error (see **Section 4.3**) occurs.

(14) Nesting is not supported for the PUBLIC BELOW...ENDP structure. (See **[Examples]**.)

(15) If the PUBLIC statement has been described in a macro, a global symbol can be declared by PUBLIC. If, however, a local symbol is declared by PUBLIC, an error (F190: Impossible to write a local symbol) occurs.

12.2 EXTRN

[Format]

```
[<label>:][Δ]EXTRNΔ<attribute>:<symbol>[,<symbol>...][;comment]
```

<attribute>: Symbol type (one of DAT, MEM, FLG, and LAB)

[Coding example]

Defining and referencing the DAT, MEM, FLG, and LAB attributes

```
; Module for referencing external symbols

            EXTRN       LAB  :  LAB1
            EXTRN       DAT  :  DATA1
            EXTRN       MEM  :  MEM03
            EXTRN       FLG  :  FLG1
             ⋮
            ADD         MEM03 , #5H
            DW          DATA1
            SUB         .FM.FLG1 SHR 4 ,#1H
            BR          LAB1

; Module for defining external symbols

            PUBLIC      BELOW               ;Public symbol declaration
            LAB1:
             ⋮
            FLG1        FLG    0.00H.0
            DATA1       DAT    1H
            ENDP

            PUBLIC      MEM03               ;Public symbol declaration
            MEM03       MEM    0.03
```

[Function]

Declares that an external module definition symbol described in the operand is to be referenced within the module.

DAT : Data type symbol
MEM : Data memory type symbol
FLG : Flag type symbol
LAB : Label type symbol

[Notes]

(1) The external reference definition made by EXTRN is required to reference a symbol defined by a program (external module) having a different assembly unit.

(2) Describe <attribute> for a symbol at the beginning of an operand field and delimit it with : (colon). Then, describe the symbol having the specified attribute. Any number of symbols can be specified within a single line; they must all have the same attribute, however.

(3) If a symbol declared by EXTRN is defined as another symbol in the same module, an error (F057: Symbol multi defined) occurs. This line is invalidated.

(4) If <attribute> and : (colon) are not described at the beginning of the operand field, an error (F037: Syntax error) occurs. A symbol is not registered as an external reference symbol. If, therefore, the symbol is referenced, an error (F058: Undefined symbol) occurs.

(5) If <symbol>, described in the operand, is not referenced, a warning (W020: Unreferenced symbol) is generated.

(6) See **Section 4.7.1** for details of DAT, MEM, FLG, and LAB.

(7) The same symbol can be declared by EXTRN two or more times.
Note, however, that if two or more identical symbols are declared, they must have the same attribute. If their attributes differ, an error (F045: Invalid type) occurs. This line is invalidated.

(8) If the operand is omitted, an error (F037: Syntax error) occurs.

(9) If the <label> description is invalid, an error (see **Section 4.3**) occurs.

(10) If a character string other than the reserved words that can be described for the type is described, an error (F037: Syntax error) occurs.

(11) If a reserved word (excluding the assembly time variables) is described in <symbol>, an error (F037: Syntax error) occurs. This line is invalidated. Therefore, the symbols described in <symbol> are not defined as external reference symbols.

(12) If an assembly time variable is described in <symbol>, an error (F166: Invalid EXTRN statement) occurs. This line is invalidated. Therefore, the symbols described in <symbol> are not defined as external reference symbols.

[MEMO]

CHAPTER 13 CONDITIONAL ASSEMBLY PSEUDO INSTRUCTIONS

This chapter explains the conditional assembly pseudo instructions that are used to evaluate expressions described as operands and thus make decisions as to whether assembly is to be executed.

The conditional assembly pseudo instructions are as follows:

- IF ... ELSE ... ENDIF
  (Assembles those instructions between IF and ELSE or between ELSE and ENDIF, depending on the result of evaluating the operand.)

- CASE ... EXIT ... OTHER ... ENDCASE
  (Assembles a block having the same numeric label as that of the result of evaluating the operand.)

- IFCHAR ... ELSE ... ENDIFC
  (Compares a character string with another string and, depending on the result, assembles those instructions between IFCHAR and ELSE or between ELSE and ENDIFC.)

- IFNCHAR ... ELSE ... ENDIFC
  (Has the same function as IFCHAR, above, except that the condition under which assembly is executed is reversed.)

- IFSTR ... ELSE ... ENDIFS
  (Checks whether a specific character string exists within a specified character string and, depending on the result, assembles those instructions between IFSTR and ELSE or between ELSE and ENDIFS.)

[Notes]

(1) If a conditional assembly pseudo instruction is described in a macro body, this pseudo instruction is interpreted when the macro is expanded. The instruction is not interpreted when the macro body is registered, however (see **Chapter 16** for details).

(2) Even if INCLUDE or a macro reference is described in a block for which assembly is skipped when a given condition is satisfied, the macro is not expanded. Also, the built-in macro instructions are not expanded, either.

(3) Even if END is described in a block for which assembly is skipped when a given condition is satisfied, the end of the source module file is assumed to be that END (see **Section 10.3** for details).

(4) Conditional assembly pseudo instructions can be nested up to 40 levels deep, including repetitive pseudo instructions and macro instructions. If the nesting level exceeds 41, an error (A035: Nesting overflow) occurs. Assembly is terminated.

(5) A block for which assembly is skipped by a conditional assembly pseudo instruction is not checked (for syntax, etc.). Therefore, no error occurs. However, nesting of the conditional assembly pseudo instruction is controlled even during skip.
If the nesting level exceeds 41 during assembly skip, an error (A035: Nesting overflow) occurs. Assembly is terminated.

(6) If the description of a label is invalid, an error occurs (see **Section 4.3** for details).

(7) For details of the relations between the conditional assembly pseudo instructions and INCLUDE pseudo instruction, see **(1)** in **Section 19.1.1**.

[Processing during assembly skip]

Nesting is controlled even during assembly skip. This is because, if IF is encountered again during skip processing for IF, the ENDIF corresponding to that IF is encountered, the next ENDIF corresponding to the IF during skip processing.

To control nesting, the nesting of conditional assembly pseudo instructions and repetitive pseudo instructions is controlled, even during skip processing. At this time, the syntax is not checked because skip processing is in progress. For ELSE, therefore, even if a label or operand is described for that ELSE, ELSE is assumed and the processing is performed accordingly.

During skip processing, the SUMMARY control instruction unconditionally skips up to ENDSUM.

(1) If IF appears during skip by IF

```
        DDD1    SET     1

        IF      DDD1=0              ;Skip processing is performed.
                :
                IF      AAA=1       ;Normally, an error occurs because no symbol
                                    ;is defined for AAA.  Because, however, the
                                    ;syntax is not checked during skip processing,
                                    ;IF is assumed and the nesting level is
                                    ;incremented by one.
                        :
                        :
                        :
LLL:            ENDIF               ;Normally, an error occurs.  However, because
                                    ;the syntax is not checked during skip
                                    ;processing, this ENDIF decrements the nesting
                                    ;level by one.

                SUMMAY,             ;Because the description up to ENDSUM is a
                Summary statement   ;summary statement, processing skips to ENDSUM.
                ENDIF               ,The description between SUMMARY and ENDSUM is
                ENDSUM              ;handled as a comment by the assembles.

        ENDIF                       ;Skip processing ends.
```

[Relations between conditional assembly pseudo instructions and a macro]

A conditional assembly pseudo instruction described in a macro body must be closed within that macro. Otherwise, an error will occur once the macro body is expanded. The conditional assembly pseudo instruction that has caused the error is forcibly closed as soon as the error occurs.

If an error occurs, one of the following messages will be output.

- F029: No ENDIF statement ... IF is not closed within a macro body.
- F030: No ENDCASE statement ... CASE is not closed within a macro body.
- F101: No ENDIFC statement ... IFCHAR is not closed within a macro body.
- F102: No ENDIFNC statement ... IFNCHAR is not closed within a macro body.
- F103: No ENDIFS statement ... IFSTR is not closed within a macro body.

(1) If only IF is described within a macro body

```
DAT1    DAT     1
MAC1            <- Macro reference
        :
        MAC2    <- Macro reference in macro
                :
                IF DAT1=0 ; Assembly is skipped.
                :
                MAC3    <-  Macro reference being skipped is not expanded.
                :
                ENDM    <-  Error (F029:  No ENDIF statement) occurs at the end of macro
                            expansion, and macro MAC2 is not processed normally.  IF is
                            unconditionally closed at ENDM, and is assembled starting from the
                            next line.
        :
        EMDM    <- Macro expansion ends normally.
        :
ENDIF           <- Error (F121:  Invalid ENDIF statement) occurs.
```

(2) If IF is encountered during skip of IF

```
DAT1    DAT     1
MAC1            <-  Macro reference
        :
        IF DAT1=0 ; Assembly is skipped.
        :
        MAC3    <-  Macro reference that is skipped is not expanded.
        :
                IF  DAT1=1  <-  This IF is skipped because skip processing is in progress.
        :
        ENDM    <-  Error (F029:  No ENDIF statement) occurs at end of macro expansion, and
                    macro MAC1 is not processed correctly.  Two IFs are unconditionally closed
                    at ENDM, and are assembled starting from the next line.
        :
ENDIF           <-  Error (F121:  Invalid ENDIF statement) occurs.
```

(3) If only an ENDIF statement is described in a macro body

```
DAT1    DAT     1
:
IF DAT1=1
:
MAC1            <-  Macro reference
        :
        ENDIF   <-  Error (F121:  Invalid ENDIF statement) occurs.
        :
        EMDM    <-  End of macro expansion
:
ENDIF           IF ends here.
```

13.1 IF ... ELSE ... ENDIF

[Format]

```
[<label>:][Δ]IFΔ<expression>[Δ][;<comment>]
                          ⋮
                     statements
                          ⋮
           [ELSE][Δ][;<comment>]
                          ⋮
                     statements
                          ⋮
           ENDIF[Δ][;<comment>]
```

[Coding example]

[Example 1] Example using ELSE

```
COND   SET    0FH
;
       IF     COND
              MOV    A, #5H          ┐
               ⋮                     │ Assembled because COND ≠ 0.
                                     ┘
       ELSE
              MOV    A, #6H          ┐
               ⋮                     │ Not assembled.
                                     ┘
       ENDIF
```

[Example 2] Example not using ELSE

```
COND   SET    0FH
;
       IF     COND
              MOV    A, #5H          ┐
              MOV    A, #4H          │ Assembled because COND ≠ 0.
               ⋮                     ┘
       ENDIF
```

[Example 3] Example of nesting

```
IF      .TYPE.TABLE1=1      ;If TABLE1 is LAB attribute, the following
                            ;is assumed:
;
        IF      ( TABLE1 AND 01FFFH) > 07FFH
                MOV     AR0 , #.DL.TABLE1 AND 0FH
                MOV     AR1 , #.DL.TABLE1 SHR 4 AND 0FH
                MOV     AR2 , #.DL.TABLE1 SHR 8 AND 0FH
                CALL    @AR
        ELSE
                CALL    TABLE1
        ENDIF
;
ELSE                        ;DAT attribute is assumed if TABLE1 is
                            ;not LAB attribute.
;
        IF      ( TABLE1 AND 01FFFH) > 07FFH
                MOV     AR0 , #TABLE1 AND 0FH
                MOV     AR1 , #TABLE1 SHR 4 AND 0FH
                MOV     AR2 , #TABLE1 SHR 8 AND 0FH
                CALL    @AR
                ELSE
                        CALL   TABLE1
                ENDIF
        ENDIF
```

[Function]

(1) IF ... ENDIF

The statements between IF and ENDIF are assembled if the result of evaluating <expression> is true (≠0).

If the result of evaluating <expression> is false (=0), the statements (IF clause) between IF and ENDIF are not assembled.

(2) IF ... ELSE ... ENDIF

If the result of evaluating <expression> is true (≠0), the statements between IF and ELSE are assembled, while those between ELSE and ENDIF are not assembled.

If the result of evaluating <expression> is false (=0), the statements (IF clause) between IF and ELSE are not assembled, but the statements (ELSE clause) between ELSE and ENDIF are assembled.

[Notes]

(1) If an error occurs in an IF statement, that IF statement becomes invalid. Consequently, an error occurs on the line of ELSE that corresponds to IF (F120: Invalid ELSE statement), and an error (F121: Invalid ENDIF statement) occurs at ENDIF.

(2) The ELSE statement does not always have to be specified.
Specify only one ELSE for an IF ... ENDIF block. If two or more ELSEs are described for one IF ... ENDIF block, the second and subsequent ELSE statements cause an error (F120: Invalid ELSE statement). The first line becomes invalid.

(3) If the description of <expression> is invalid, an error occurs (for details, see **Section 4.7.1**). That line becomes invalid.

(4) Describe a symbol that has already been defined as the operand <expression> of an IF statement. If a symbol that is not defined in the same module or a forward reference symbol is described, an error (F058: Undefined symbol) occurs. That line becomes invalid.

(5) The result of evaluating the operand <expression> of an IF statement must be of data type. If the evaluation result is of other than data type, an error (F045: Invalid type) occurs. That line becomes invalid.

(6) If there is no ENDIF specified corresponding to IF, an error (F029: No ENDIF statement) occurs, either at the END statement or at the end of the file.

(7) If only ENDIF is described without IF, an error (F121: Invalid ENDIF statement) occurs. That line becomes invalid. If only ELSE is described, an error (F120: Invalid ELSE statement) also occurs. That line becomes invalid.

(8) If a label or operand is described on the line of ELSE or ENDIF, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If an external definition symbol is described as <expression>, an error (F150: Impossible to write the external symbol) occurs. The IF statement becomes invalid.

(10) If an IF or ELSE statement is described in a macro body, the clause must be always closed within the macro body.

(11) If the operand <expression> of an IF statement is omitted, or two or more <expression> are described, an error (F037: Syntax error) occurs. The IF statement becomes invalid.

13.2 CASE ... EXIT ... OTHER ... ENDCASE

[Format]

```
[<label>:]       [Δ]CASEΔ<expression>[Δ][;<comment>]

<numeric-label>: [Δ][;<comment>]
[<label>:]       [Δ]statements
                 [Δ][EXIT]

<numeric-label>: [Δ][;<comment>]
[<label>:]       [Δ]statements
                 [Δ][EXIT]

                     ⋮
[:<data-type-symbol>:]
                     ⋮
[OTHER:][Δ][;<comment>]
[<label>:]statements

                   ENDCASE[Δ][;<comment>]
```

[Coding example]

(1) Example of CASE and EXIT

```
        CASE            N       ;  (0FH)=(0FH)+N
        ;
3:                              ;
        MOV     MEM01,  #1H     ;  Range to be assembled where N=3.
        ADD     MEM01,  #2H     ;
00B:                            ;
        CALL    LAB1            ;  Range to be assembled where N=00B.
        SUB     MEM02,  #5H     ;
        EXIT                    ;
0FH:                            ;
        MOV     MEM03,  #4H     ;  Range to be assembled where N=0FH.
        EXIT                    ;
OTHER:                          ;
        SUB     MEM03,  #5H     ;  Range to be assemble where N≠3,
ENDCASE                         ;  N≠00B, and N≠0FH.
```

[Function]

Assembles those statements between <numeric-label>, having the same value as the result of evaluating <expression> described as the operand of the CASE statement, and the ENDCASE statement. A "numeric label" is a numeral expressed as a 7-bit ASCII code followed by a colon (:).

If the EXIT statement appears during assembly, processing is stopped, and execution exits from the innermost CASE block.

If no numeric label equivalent to the value of <expression> exists, the block between OTHER and ENDCASE is assembled. Note, however, that the OTHER block must be described after the numeric label block. If the OTHER block is omitted, and no numeric label equivalent to the value of <expression> exists, all statements between CASE and ENDCASE are skipped and not assembled.

[Notes]

(1) Statements other than a comment statement cannot be described on the numeric label line. If a mnemonic or operand is described on the numeric label line, an error (F069: Invalid CASE LABEL) occurs. That line becomes invalid.

(2) The value of the numeric label must be an integer in the range of $0 \leq x \leq 0FFFFFFFFH$. If a negative value or a value outside this range is described, an error (F164: The constant is over 32 bits) occurs. That line becomes invalid. Describe the numeric label in binary, octal, decimal, or hexadecimal.

(3) For numeric labels numerals need not to be described in ascending or descending order.

(4) If the same numeric label is described two or more times in a single CASE block, only that which appears first becomes valid. The second and subsequent specifications are ignored.

(5) A label other than a numeric label can be described in the CASE ... ENDCASE block.

(6) If a label or operand is described for ENDCASE or EXIT, an error (F037: Syntax error) occurs. That line becomes invalid.

(7) If another OTHER is described in the CASE ... ENDCASE block, an error (F123: Invalid OTHER statement) occurs. That line becomes invalid.

(8) If a numeric label is described after an OTHER statement, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) Only one OTHER statement can be described in a CASE ... ENDCASE block. If two or more OTHER statements are described, the second and subsequent statements cause an error (F123: Invalid OTHER statement). That line becomes invalid.

(10) The EXIT statement may be described as many times as required.

(11) If there is no ENDCASE corresponding to CASE, an error (F030: No ENDCASE statement) occurs, either at the END statement or at the end of the file.

(12) If ENDCASE is described before CASE, an error (F122: Invalid ENDCASE statement) occurs. That line becomes invalid. If EXIT is described, an error (F080: Invalid EXIT statement) occurs. Similarly, if OTHER is described, an error (F123: Invalid OTHER statement) occurs. In both cases, the line becomes invalid.

(13) A data type symbol can be used for <numeric-label> by describing ":" first.

Coding format `:<data-type-symbol>:`

If a symbol of other than data type is described at this time, an error (F045: Invalid type) occurs. That line becomes invalid.

(14) If an undefined symbol or forward reference symbol is described for <expression>, an error (F058: Undefined symbol) occurs. The CASE statement becomes invalid.

(15) If an external definition symbol is described for <expression>, an error (F150: Impossible to write the external symbol) occurs. The CASE statement becomes invalid.

(16) If any error occurs in <expression> (see **Section 4.7.1**), the CASE statement becomes invalid.

(17) If any error occurs in the CASE statement, the CASE statement becomes invalid. At this time, each statement corresponding to this CASE statement is assumed to be as follows:

- Line containing numeric label for CASE statement -> F069: Invalid CASE LABEL
- OTHER statement -> F123: Invalid OTHER statement
- ENDCASE statement -> F122: Invalid ENDCASE statement
- EXIT statement -> F080: Invalid EXIT statement

(18) If no operand is specified for a CASE statement, or two or more operands are described, an error (F037: Syntax error) occurs, and that CASE statement becomes invalid.

(19) No error occurs even if no numeric label or OTHER is described between CASE and ENDCASE. In this case, however, all statements between CASE and ENDCASE are skipped and not assembled.

```
        CASE    0
        IF  1           ; This IF is skipped.

        EXIT
        ELSE            ; This ELSE is also skipped.
        ENDCASE
```

(20) The result of evaluating the operand <expression> of a CASE statement must be of data type. Otherwise, an error (F045: Invalid type) occurs. That line becomes invalid.

(21) Relation between CASE and IF

The nesting level does not increase even if a pseudo instruction related to nesting, such as IF, is encountered during skip processing in a CASE block. During skip processing for the CASE block, a search is made for the corresponding numeric label and CASE pseudo instruction. If a corresponding numeric label is found, assembly is started from that label. If a CASE pseudo instruction is found, that CASE block is skipped.

```
        A       SET     10
        B       SET     10

        CASE    A
0:
        IF B=10                 ;Nesting level does not increase during skip
                                ;processing for CASE block.
                NOP
        ELSE
                NOP
        ENDIF
5:
                CASE    B       ;Skipped up to ENDCASE.  Therefore,
                                ;a numeric label corresponding to this CASE
                                ;is also skipped.

        0:
        5:
        10:
                ENDCASE
10:                             ;Assembly starts from here.
        IF B=10
                NOP
        ELSE
                NOP
        ENDIF
        ENDCASE
```

13.3 IFCHAR ... ELSE ... ENDIFC

[Format]

```
[<label:>][Δ]IFCHARΔ<string-1>,<string-2>[,<start-character-position>,<end-character-position>][;<comment>]
            ⋮
       statements
            ⋮
   [ELSE][Δ][;<comment>]
            ⋮
       statements
            ⋮
   ENDIFC[Δ][;<comment>]
```

[Coding Examples]

[Example 1] When a character constant is specified

```
      IFCHAR ' ABC' , ' BBB' , 1, 1
      ⋮
      ELSE
      ⋮
      ENDIFC
```

In this case, the statements between IFCHAR and ELSE are assembled, even if a character constant is specified, because the first character of the constant is a quotation mark (').

[Example 2] If both string 1 and string 2 are omitted

```
      IFCHAR , , 1 , 1
      ⋮
      ELSE
      ⋮
      ENDIFC
```

In this case, the statements between ELSE and ENDIFC are always assembled because there are no character strings to be compared.

[Example 3] If string 1, string 2, the character start position, and the character end position are omitted

```
IFCHAR ,
```

An error (F037: Syntax error) occurs.

[Function]

IFCHAR, ELSE, and ENDIFC are conditional pseudo instructions that compare string 1 and string 2, described as operands. If string 1 and string 2 coincide (i.e., if the result of evaluating the operands is "true"), the statements between IFCHAR and ENDIFC (or ELSE) are expanded. The block is expanded depending on whether the result of the evaluation is true or false, in accordance with IF, ELSE, and ENDIF. The characters to be compared are specified by the start and end character positions. If no start or end character position is specified, all characters in the character strings are compared.

[Notes]

(1) String 1 and string 2 may consist of symbols, expressions, numeric values, or characters. Special characters other than a comma (,) can also be described. The system differentiates between uppercase and lowercase characters.
If a character constant is described by using quotation marks, the quotation marks are regarded as being part of the character string.

Example
```
IFCHAR ' AAA',' BBB', 1, 1
```

In this example, whether the first character of string ' AAA' is the same as the first character of string ' BBB' is checked. For both strings, the first character is a quotation mark (').

(2) The first character of a character string is assumed to be the first character. If "0" is described as the start/end character position, therefore, an error (F044: Invalid value) occurs. That line becomes invalid.

(3) As the start/end character, describe a numeric constant as a binary, octal, decimal, or hexadecimal number in the range of $1 \leq n \leq 253$. If any other item (such as an expression) is described, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) If the value of the start character position is greater than that of the end character position, an error (F044: Invalid value) occurs. That line becomes invalid.

(5) Even if the value of the start or end character position is greater than string 1 or 2, no error occurs.

- If the value of the start character position is greater than that of the character string
  The statements between IFCHAR and ELSE or between IFCHAR and ENDIFC are not assembled.

- If the value of the end character position is greater than that of the character string
  The character string is checked from the specified start character position, up to the end.

(6) If ENDIFC corresponding to IFCHAR is missing, an error (F101: No ENDIFC statement) occurs, either in the END statement or at the end of the file.

(7) If ENDIFC is described before IFCHAR, an error (F124: Invalid ENDIFC statement) occurs. That line becomes invalid. Similarly, if ELSE is described, an error (F120: Invalid ELSE statement) occurs. That line becomes invalid.

(8) If a label is described for ELSE or ENDIFC, or if a mnemonic or operand is described, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If no operand or more than five operands are specified for an IFCHAR statement, an error (F037: Syntax error) occurs. The IFCHAR statement becomes invalid.

Remark IFCHAR and IFNCHAR, explained in the next section, are pseudo instructions that are used for macro definition. These instructions enable the control of conditional assembly processing in a macro based on a macro parameter passed by a macro call statement.

[Example] Example of extended addition instruction

In the following example, it is assumed that the macro is never expanded if 0 is specified as the operand of the REPT pseudo instruction.

```
;
; In the following example, the symbol representing the data memory attribute is
; described as A, while the symbol representing the data memory attribute or data
; attribute starting from "#" is described as B.
; Note  This example does not correspond to the symbol of the data memory
;       attribute defined by the NIBBLEnV pseudo instruction.

MEM_TYPE        DAT     3           ; Defines the return value for the data memory
                                    ; attribute by .TYPE. function

; Macro definition

ADD_EX       MACRO    A,B

IF .TYPE.A  <> MEM_TYPE
  ZZZERROR '1ST OPERAND TYPE ERROR'
ELSE
  IFCHAR #,B,1,1                    ; If second operand starts with #
;
    ADD A, B AND 0FH                ; Addition of least significant digit
;
                COUNT SET1          ; Initial setting of macro loop counter
    REPT A SHR 12                   ; The following addition is repeated by
                                    ; (number of nibbles - 1)
      ADDC  A+COUNT, B SHR (COUNT*4)  AND 0FH
                COUNT SET COUNT+1   ; Updates loop counter
    ENDR
;
  ELSE                              ; If second operand does not start with #, bit
                                    ; length is determined by first operand.
    IF .TYPE.B <> MEM_TYPE
      ZZZERROR '2ND OPERAND TYPE ERROR'
    ELSE
        ADD  A, B                   ; Addition of least significant digit
;
                COUNT SET1          ; Initial setting of macro loop counter

      REPT A SHR 12                 ; The following addition is repeated as many
                                    ; times as number of nibbles - 1.
        IF COUNT =< (B SHR 12)  ;
           ADDC  A+COUNT, B+COUNT
        ELSE                        ;If the second operand is shorter than the
                                    ; first operand
           ADDC  A+COUNT,#0
        ENDIF
                COUNT SET COUNT+1   ; Updates loop counter
        ENDR
    ENDIF
  ENDIFC
ENDIF
ENDM
```

```
; This macro is called as follows.

; Symbol (variable) definition
; Note  A symbol is defined for the lower bit of a variable in this example.

MEMORY3         NIBBLE3 0.10H
REGISTER4       NIBBLE  0.0H
DATA            DAT      123H

    ; Call

ADD_EX      MEMORY3, #DATA          ; <1>
ADD_EX      REGISTER3, MEMORY       ; <2>

    ; Expansion of <1>

ADD     MEMORY3, #DATA  AND 0FH
ADDC    MEMORY3+COUNT,  #DATA SHR (COUNT*4) AND 0FH
ADDC    MEMORY3+COUNT,  #DATA SHR (COUNT*4) AND 0FH

    ; Expansion of <2>

ADD     REGISTER4, MEMORY3
ADDC    REGISTER4+COUNT, MEMORY3+COUNT
ADDC    REGISTER4+COUNT, MEMORY3+COUNT
ADDC    REGISTER4+COUNT, #0
```

13.4 IFNCHAR ... ELSE ... ENDIFNC

[Format]

```
[<label>:][Δ]IFNCHARΔ<string-1>,<string-2>[,<start-character-position>, <end-character-position>][;<comment>]
                ⋮
          statements
                ⋮
     [ELSE][Δ][;<comment>]
                ⋮
          statements
                ⋮
     ENDIFNC[Δ][;<comment>]
```

[Function]

IFNCHAR, ELSE, and ENDIFNC are conditional pseudo instructions that compare string 1 with string 2, both of which are described as operands. If string 1 and string 2 are found to differ (i.e., if the result of evaluating the operands is "false"), the statements between IFNCHAR and ENDIFNC (or ELSE) are expanded. The block is expanded, depending on whether the result of the evaluation is true or false, in accordance with IF, ELSE, and ENDIF.

The characters constituting a string to be compared are specified by means of the start and end character positions. If no start or end character position is specified, all characters in the character strings are compared.

[Notes]

(1) String 1 and string 2 may consist of symbols, expressions, numeric values, or characters. Special characters other than a comma (,) can also be described. The system differentiates between uppercase and lowercase characters.

If a character constant is described using quotation marks, the quotation marks are regarded as being part of the character string.

[Example]

```
IFNCHAR ' AAA',' BBB',1,1
```

In this example, whether the first character of string ' AAA' is the same as the first character of string ' BBB' is checked. The first character of both strings is a quotation mark (').

(2) The first character of a character string is assumed to be the first character. If "0" is described as the start/end character, therefore, an error (F044: Invalid value) occurs. That line becomes invalid.

(3) As the start/end character, describe a numeric constant as a binary, octal, decimal, or hexadecimal number in the range of $1 \leq n \leq 253$. If any other item (such as expression) is described, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) If the value of the start character is greater than that of the end character, an error (F044: Invalid value) occurs. That line becomes invalid.

(5) Even if the value of the start character or end character is greater than string 1 or 2, no error occurs.

- If the value of the start character is greater than that of the character string
  The statements between IFNCHAR and ELSE or between IFNCHAR and ENDIFNC are not assembled.

- If the value of the end character position is greater than that of the character string
  The character string is checked from the specified start character position, up to the end.

(6) If ENDIFNC corresponding to IFNCHAR is missing, an error (F102: No ENDIFNC statement) occurs, either in the END statement or at the end of the file.

(7) If ENDIFNC is described before IFNCHAR, an error (F125: Invalid ENDIFNC statement) occurs. That line becomes invalid. Similarly, if ELSE is described, an error (F120: Invalid ELSE statement) occurs. That line becomes invalid.

(8) If a label is described for ELSE or ENDIFNC, or if a mnemonic or operand is described, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If no operand or more than five operands are specified for an IFNCHAR statement, an error (F037: Syntax error) occurs, and the IFNCHAR statement becomes invalid.

13.5 IFSTR ... ELSE ... ENDIFS

[Format]

```
[<label>:][Δ]IFSTRΔ<string-1>,<string-2>[,<string-3>...][Δ][;<comment>]
                ⋮
          statements 1
                ⋮
    [ELSE][Δ][;<comment>]
                ⋮
          statements 2
                ⋮
    ENDIFS[Δ][;<comment>]
```

[Function]

Selects whether statements 1 or statements 2 are assembled depending on whether string 1 coincides with string 2 and the subsequent strings.
If string 1 coincides with any of string 2 or the subsequent strings, the instructions described as statement 1 are assembled. If string 1 does not coincide with any of string 2 and the subsequent strings, the instructions described as statement 2 are assembled. If, however, the ELSE statement is not specified, all statements between IFSTR and ENDIFS are skipped and not assembled.
As a string, symbols, expressions, and numeric values may be described. To describe a string, it need not be enclosed in quotation marks ('). If a string is enclosed by quotation marks, those quotation marks are regarded as being part of the string. Special characters, other than a comma (,), may be described.

[Notes]

(1) IFSTR and ENDIFS statements must always be specified as a pair within a given level.
(2) If only the ELSE statement is described, an error (F120: Invalid ELSE statement) occurs. That line becomes invalid. Similarly, if only the ENDIFS statement is described, an error (F126: Invalid ENDIFS statement) occurs. That line becomes invalid.

(3) The number of characters that can be described for <string-2> and subsequent strings is equal to the number of characters that can be described on one line (253).

(4) If no ENDIFS statement is specified as the partner to an IFSTR statement, an error (F103: No ENDIFS statement) occurs, either in the END statement or at the end of the file.

(5) If ENDIFS is described before IFSTR, an error (F126: Invalid ENDIFS statement) occurs. That line becomes invalid.

(6) If ELSE is described before IFSTR, an error (F120: Invalid ELSE statement) occurs. That line becomes invalid.

(7) The system distinguishes between uppercase and lowercase characters when character strings are compared.

(8) If a label, mnemonic, or operand is described on the ELSE or ENDIFS line, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If the operand of the IFSTR statement is omitted, an error (F037: Syntax error) occurs. The IFSTR statement becomes invalid.

[Example]

[Example 1]

```
STRCMP    MACRO       STR1     ;
  IFSTR     STR1,DAT1,MEM1     ; Judges whether string of macro parameter
            :                  ; STR1 is DAT1 or MEM1.
            :                  ; If STR1 is DAT1 or MEM1, expands macro.
            :

  ELSE
            :                  ; If STR1 is neither DAT1 nor MEM1, expands
            :                  ; macro.
            :
  ENDIFS
ENDM
```

[Example 2]

```
  IFSTR   'ABCD',ABCD
          NOP
  ELSE
          NOP
  ENDIFS
```

In Example 2 above, the statements between ELSE and ENDIFS are assembled (because quotation marks (') are regarded as being part of a character string, strings 'ABCD' and ABCD do no coincide).

[MEMO]

CHAPTER 14 REPETITIVE PSEUDO INSTRUCTIONS

This chapter explains the repetitive pseudo instructions.
The repetitive pseudo instructions are as follows:

- IRP ... ENDR (Expands a specified block while replacing formal parameters with actual parameters.)
- REPT ... ENDR (Expands a specified block a specified number of times.)

[Notes]

(1) The symbols defined in IRP and REPT are global symbols. Therefore, they can be referenced from outside IRP and REPT.

(2) A macro cannot be defined in IRP and REPT. If defined, an error (F183: Invalid MACRO place) occurs on the line on which the macro pseudo instruction is described.

[Relation between repetitive pseudo instructions and macro]
When describing a repetitive pseudo instruction in a macro body, the instruction must be closed within the macro body. Otherwise, an error (F031: No ENDR statement) occurs when the macro body is expanded.
The repetitive pseudo instruction that is responsible for the error is forcibly closed as soon as the error occurs.

(1) If only the IRP statement is described in the macro body

```
DAT1    DAT     1
MAC1            <- Macro reference
          :
        MAC2    <- Macro reference in macro
                    :
                IRP X,1,2,3
                    :
                ENDM        <- Error (F031: No ENDR statement) occurs when macro has
                               been expanded, but macro MAC2 is not correctly processed.
                               IRP is unconditionally closed by ENDM and assembly is
                               executed normally, starting from the next line.
          :
        ENDM                <- Macro expansion is completed normally.
          :
ENDR                <- Error (F041: Invalid ENDR statement) occurs.
```

(2) If REPT is described during the processing of IF

```
DAT1   DAT    1
MAC1           <-  Macro reference
        :
        IF  DAT1=1      ; Statements between IF and ENDIF are assembled.
        :
              REPT5
        :     :
        ENDM   <-  Error (F031:  No ENDR statement) occurs when macro has been
                   expanded, and macro MAC1 is not processed correctly.
                   REPT and IF are unconditionally closed by ENDM and assembly is
                   executed normally, starting from the next line.
        :
ENDIF          <-  Error (F121:  Invalid ENDIF statement) occurs.
```

14.1 IRP ... ENDR

[Format]

```
[<label>:][Δ]IRPΔ<formal-parameter>,<actual-parameter-list>[;<comment>]
                ⋮
        [EXITR][;<comment>]
                ⋮
        ENDR[;<comment>]
```

[Function]

Repeatedly expands the block enclosed between IRP and ENDR, as many times as the number of actual parameters specified for IRP (data described in the actual parameter list). When the IRP ... ENDR block is expanded, the formal parameters described in the block are replaced by the actual parameters. Each time the block is expanded, the actual parameters change in the sequence described in the actual parameter list.

[Notes]

(1) The character string described as the first operand is assumed to be a formal parameter. A formal parameter consists of characters (complying with the symbol coding rules). (The length of both the formal and actual parameters is limited by the 253 characters-per-line maximum.) The coding format of the actual parameter list, described as the second and subsequent operands, is the same as that used when a macro is referenced.

(2) If no actual parameter is described, an error (F037: Syntax error) occurs. That line becomes invalid.

(3) If EXITR is encountered while the IRP ... ENDR block is being expanded, the innermost repeat processing is terminated (for details, see **Section 14.3**).

(4) The IRP ... ENDR block can be nested up to 40 levels deep, in combination with other repetitive and conditional assembly pseudo instructions and macro pseudo instruction blocks. If the nesting level exceeds 41, however, an error (A035: Nesting overflow) occurs. The assembly processing is terminated.

(5) IRP must be terminated with ENDR. If ENDR is not described, an error (F031: No ENDR statement) occurs, either in the END statement or at the end of the file.

(6) If IRP is not described before ENDR, an error (F041: Invalid ENDR statement) occurs. That line becomes invalid.

(7) A macro body must be enclosed between IRP and ENDR. If only one of either IRP or ENDR is described, an error occurs at ENDM when the macro is expanded (F031: No ENDR statement if only IRP is described, and F041: Invalid ENDR statement if only ENDR is described).

(8) If a label or operand is described on the description line for ENDR, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If the operand of IRP is omitted, an error (F037: Syntax error) occurs. IRP becomes invalid.

(10) If <label> is described erroneously, an error occurs (see **Section 4.3**).

(11) If <formal-parameter> is omitted, an error (F037: Syntax error) occurs. That line becomes invalid.

(12) If any error occurs during IRP expansion, the description line number for ENDR is output to the log file.

14.2 REPT ... ENDR

[Format]

```
[<label>:][Δ]REPTΔ<expression>[;<comment>]
                ⋮
           [EXITR][;<comment>]
                ⋮
           ENDR[;<comment>]
```

[Function]

Repeatedly expands the statements enclosed between REPT and ENDR as many times as the value specified for <expression>. If EXITR is encountered between REPT and ENDR, the innermost REPT processing is terminated.
The evaluation value of <expression> is 32 bits long.

[Notes]

(1) The REPT ... ENDR block can be nested up to 40 levels deep, in combination with other repetitive and conditional assembly pseudo instructions and macro pseudo instruction blocks. If the nesting level exceeds 41, however, an error (A035: Nesting overflow) occurs. Assembly processing is terminated.

(2) If the description of <expression> is invalid, an error occurs (for details, see **Section 4.7.1**). That line becomes invalid.

(3) To describe a symbol for <expression>, that symbol must have already been defined. If a forward reference symbol or undefined symbol is described, an error (F058: Undefined symbol) occurs. The REPT statement becomes invalid.

(4) If an external definition symbol is described for <expression>, an error (F150: Impossible to write the external symbol) occurs. The REPT statement becomes invalid.

(5) An REPT pseudo instruction must be terminated with ENDR. If ENDR is not described, an error (F031: No ENDR statement) occurs, either in the END statement or at the end of the file.

(6) If an REPT pseudo instruction is not described before an ENDR pseudo instruction, an error (F041: Invalid ENDR statement) occurs. That line becomes invalid.

(7) A macro body must be enclosed between REPT and ENDR. If only one of either REPT or ENDR is described, an error occurs at ENDM when the macro is expanded (F031: No ENDR statement if only REPT is described, and F041: Invalid ENDR statement if only ENDR is described).

(8) If a label or operand is described on the description line for ENDR, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If the operand of REPT statement is omitted, or if two or more operands are described, an error (F037: Syntax error) occurs. The REPT statement becomes invalid.

(10) If <label> is described erroneously, an error occurs (see **Section 4.3**).

(11) If the evaluation value of <expression> is 0, the statements enclosed by REPT and ENDR are not expanded.

(12) Only a data type expression can be described for <expression>. If an expression of any other type is described, an error (F045: Invalid type) occurs. That line becomes invalid.

(13) If any error occurs during REPT expansion, the description line number of ENDR is output to the log file.

[Example]

```
LOC.  OBJ.  M   I   STATEMENT

                        D_MOJI      SET     0F00H
                        ;
                                    REPT    3
                        D_MOJI      SET     D_MOJI + 1
                                    DB      D_MOJI
                        ENDR
                        D_MOJI      SET     D_MOJI + 1
      0F01                          DB      D_MOJI
                        D_MOJI      SET     D_MOJI + 1
      0F02                          DB      D_MOJI
                        D_MOJI      SET     D_MOJI + 1
      0F03                          DB      D_MOJI
```

14.3 EXITR

[Format]

```
[<label>:][Δ]EXITR[Δ][;comment]
```

[Function]

If an EXITR pseudo instruction is encountered while REPT/IRP is being expanded, the innermost REPT/IRP processing is aborted, and assembly is resumed starting from the statement subsequent to ENDR.

[Notes]

(1) If EXITR is described at a location other than the REPT/IRP ... ENDR block, an error (F042: Invalid EXITR statement) occurs. That line becomes invalid.

(2) If an operand is described for the EXITR statement, an error (F037: Syntax error) occurs. That line becomes invalid.

(3) If the description of <label> is invalid, an error occurs (see **Section 4.3**).

[Example]

(1)

```
IRP             X, P1, P2, P3
;
IF              X > 05H
        ADD             MEM00,#X&H
ELSE
        EXITR
ENDIF
ENDR
```

The statements between IRP and ENDR are expanded, and parameter X is sequentially replaced by P1, P2, and P3. In this example, if the condition is false, the statements in the IF block following ELSE are assembled. IRP processing is aborted at EXITR.

(2) If EXITR exists in a macro

```
IRP            X, P1, P2, P3
;
ABC           ; Macro reference
     ⋮
        IF              X > 05H
             ADD               MEM00,#X&H
        ELSE
             EXITR <- Terminates ABC macro expansion and aborts IRP processing.
        ENDIF
ENDM
ENDR
```

A macro is referenced in IRP. If EXITR is encountered during macro expansion, all statements up to ENDR are skipped.

CHAPTER 15 MESSAGE CREATION PSEUDO INSTRUCTIONS

RA17K supports two pseudo instructions that are used to create messages. During assembly, created messages are output to the console in real time. In addition to these messages, the numbers of the lines on which the messages are described are also output to the console.

- ZZZERROR pseudo instruction (Outputs and counts error messages.)
- ZZZMSG (Outputs messages.)

[Notes]

(1) If the description of <label> is invalid, an error (see **Section 4.3**) occurs.

(2) If an operand is omitted, or if two or more operands are described, an error (F037: Syntax error) occurs.

15.1 ZZZERROR PSEUDO INSTRUCTION

[Format]

```
[<label>:][Δ]ZZZERRORΔ'<string>'[;<comment>]
```

[Function]

Transfers the character string, described as the operand and enclosed in single quotation marks ('), to the OS as an error message. The character string is output to the console and log file during assembly, in the same way as an error message output by the system.
Also, these messages are counted as part of the total number of errors output during assembly.

[Notes]

(1) When a message created by the ZZZERROR pseudo instruction is output, the line number of the source is also output, in the same way as for messages created by the system. However, messages output by ZZZERROR have no error numbers and, therefore, no error number is output.

(2) The sequence in which messages are created by the ZZZERROR pseudo instruction does not correspond to the line numbers of the source program.

(3) <string> specified for ZZZERROR must always be enclosed in single quotation marks. Otherwise, an error (F037: Syntax error) occurs. That line becomes invalid.

[Example]

Mask option definition macro

```
OPTCK   MACRO   XIP,CKP
        IF      ( XIP > 2 ) OR ( XIP = 0)
                ZZZERROR        'Invalid value for SYSTEM CLOCK'
        ELSE
                ZZZOPT  5,0,XIP
                IF      ( CKP > 2) or ( CKP = 0)
                        ZZZERROR' Invalid value for SYSTEM CLOCK at reset time'
                ELSE
                        IF  (CKP = 10B) AND (XIP = 01B)
                                    ZZZERROR' Invalid selection : NOXT,INITCKXT'
                        ELSE
                                    ZZZOPT  2,0,CKP
                        ENDIF
                ENDIF
        ENDIF
ENDM
```

[Output example]

Assume that the following line exists in module TEST.ASM.

```
Line No.     Mnemonic
100          ZZZERROR 'Error occurs.'
```

When this module is assembled, the following message is output to the screen and log file.

```
TEST.ASM(100) error     : Error occurs.
```

15.2 ZZZMSG PSEUDO INSTRUCTION

[Format]

```
[<label>:][Δ]ZZZMSGΔ'<string>'[;<comment>]
```

[Function]

Transfers the character string, described as the operand and enclosed in single quotation marks ('), to the OS as a message. The character string is output to the console and log file during assembly. However, the total number of errors and warnings output at the end of assembly is not counted.

[Notes]

(1) When the message created by the ZZZMSG pseudo instruction is output, the line number of the source is also output, in the same manner as messages created by the system. At this time, the type of the message is not output.

(2) The sequence in which messages are created by the ZZZMSG pseudo instruction does not correspond to the line numbers of the source program.

(3) <string> specified for ZZZMSG must always be enclosed in single quotation marks. Otherwise, an error (F037: Syntax error) occurs. That line becomes invalid.

[Example]

```
MESSAGE_FOR_IRQ_FLAG    MACRO   FLG1
;
        IF      FLG1 SHR 4 AND 0BEH <> 0
                ZZZMSG          'CAUTION! Unexpected IRQ may be canceled.'
        ENDIF
;
        SET1            FLG1
ENDM
```

[Output example]

Assume that the following line exists in module TEST.ASM.

```
Line No.     Mnemonic
150          ZZZMSG 'This is a pseudo instruction to output a message.'
```

When this module is assembled, the following message is output to the screen and log file.

```
TEST.ASM(150)    : This is a pseudo instruction to output a message.
```

CHAPTER 16 MACRO PSEUDO INSTRUCTIONS

This chapter explains pseudo instructions, macro parameters, and operators, and how to define and reference macros.

A macro consists of a series of procedure statements (macro body) in a source program, to which a name is assigned. The macro name and procedure statements can be arbitrarily defined by the user. A macro can be used simply by describing the corresponding macro name in the source program (macro reference). During assembly, the assembler replaces the macro name with the defined statements. Thus, a macro must be defined before it can be referenced. A defined macro can be used as many times as necessary within a source module file. A macro can pass parameters and the destinations to which the parameters are passed can be changed each time the macro is referenced.

The use of macros can also improve the legibility of a program by assigning a name that is indicative of the contents of the procedure to a series of blocks.

If identical processing were to be described repeatedly, the overall flow of a program would be difficult to understand. By calling a macro instead the coding can be simplified considerably. By defining, as a macro, a previously created procedure, the stable operation of which has been confirmed, the macro can be used in the same way as the statements. In this case, an independent file containing only the macro definition statement is created, that file being read by the INCLUDE statement at the start of execution of the source program. This technique is particularly convenient for creating libraries.

A macro can be used more efficiently when used in combination with a conditional assembly pseudo instruction (see **Chapter 13**).

Next, the definition, reference, and expansion of a macro are explained by using a simple example.

Suppose a macro, named SHIFTR, shifts the contents of a register 1 bit to the right and inserts 0 into the most significant bit position. This macro is defined, referenced, and expanded in a program as follows:

[Macro definition]

Macro definition involves allocating a series of instruction statements and pseudo instructions to a macro name.

In this example, the SHIFTR macro is defined.

```
 LABEL      MNEMONIC     OPERAND        COMMENT
SHIFTR      MACRO                       ;MACRO DEFINITION
            RORC         MEM7F
            AND          MEM7F,#7FH
ENDM
```

To the SHIFTR macro, two instructions are allocated, as shown below. The macro name must not duplicate the name of a symbol or reserved word. If an existing name is specified for the macro, an error (F057: Symbol multi defined) occurs. That line becomes invalid.

```
        RORC        MEM7F
        AND         MEM7F,#7FH
```

[Macro reference]

After macro definition, a macro can be referenced from any point within the same source module, as many times as required.

A macro reference is described in the mnemonic field of a statement.

```
 LABEL      MNEMONIC     OPERAND      COMMENT

            SHIFTR                      ;MACRO REFERENCE
```

[Macro expansion]

When a macro is referenced, the assembler expands the series of instruction statements allocated to that macro, in the defined sequence. Simply by describing a macro name in the source program, the same result as that produced when the defined instructions are described can be obtained.

```
 LABEL      MNEMONIC     OPERAND      COMMENT

            RORC        MEM7F
            AND         MEM7F,#7FH
```

The following macro pseudo instructions are supported.

- MACRO (Defines a macro name and declares the beginning of a macro body.)
- ENDM (Declares the end of a macro body.)
- GLOBAL (Declares that a symbol defined within a macro can be referenced from outside the macro.)
- PURGE (Deletes a macro that is no longer required.)

The use of these macro pseudo instructions and the referencing of a macro are described in the following sections.

16.1 DEFINING A MACRO

16.1.1 MACRO and ENDM (MACRO Definition and END of Macro)

[Format]

```
<macro-name>ΔMACRO[Δ<formal-parameter-list>][Δ][;<comment>]
                ⋮
                <macro-body (Macro body)>[Δ][;][;<comment>]
                ⋮
                ENDM[Δ][;<comment>]
```

[Function]

Allocates the name (macro name) described in the symbol field to a series of statements (macro body) between the MACRO and ENDM statements. To reference this macro body, describe a macro name and any parameters that may be necessary (for an explanation of the use of parameters, see **Section 16.5**).

[Notes]

(1) The macro body consists of a "symbol," "instructions," "pseudo instructions" (excluding MACRO and ENDM), "comments," and a "macro name" (to reference other macros).

(2) A comment statement prefixed with two semicolons (;;) in the macro body relates to the macro definition. This comment is not expanded when the macro is expanded.

(3) formal-parameter-list

- **<1>** A formal parameter consists of characters (complying with the symbol coding rules). Two or more parameters can be specified, delimited by a comma (,) (the length and number of the formal parameters must not exceed 253 characters per line).
- **<2>** A formal parameter is valid only in a macro body.
- **<3>** A formal parameter that is described in a macro body is replaced by the character string (including when a character constant is used) of that actual parameter described in the operand field when the macro is referenced.
- **<4>** No error occurs even if a reserved word is described as a formal parameter. The formal parameter is interpreted as a simple character string, and is replaced by the actual parameter when the macro is expanded.

(4) If an error occurs on the line of the MACRO statement, that line becomes invalid. Therefore, the macro body described starting from the next line is not registered, instead being interpreted as ordinary instructions and processed accordingly.

(5) If there is no ENDM corresponding to MACRO, an error (F032: No ENDM statement) occurs for the END statement or at the end of the file.

(6) If ENDM is encountered before MACRO, an error (F043: Invalid ENDM statement) occurs. That line becomes invalid.

(7) When a macro is defined, the syntax of the macro body is not checked. The syntax is checked when the macro is expanded.

(8) If a label, mnemonic, or operand is described for an ENDM statement, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If a previously defined symbol is specified as <macro-name>, an error (F057: Symbol multi defined) occurs. That line becomes invalid.

(10) If a reserved word is described for <macro-name>, an error (F037: Syntax error) occurs. That line becomes invalid.

(11) If <macro-name> is omitted, an error (F037: Syntax error) occurs. That line becomes invalid.

* (12) The maximum supported macro body size is 64K bytes. If this is exceeded, an error (F148: Macro body is over 64 K bytes) occurs. The macro is not registered (all statements up to ENDM are skipped).

(13) When a macro is defined, the character strings up to ENDM are registered as the macro body. At this time, these character strings are not checked (for syntax), the character strings up to ENDM being unconditionally registered as a macro body.
The MACRO and END character strings, however, are checked and processed.

- **If MACRO exists in a macro body**
  Because no other macro can be defined in a macro, if a MACRO statement is described in a macro body, an error (F183: Invalid MACRO place) occurs. If an error occurs in a macro body, that macro is not registered.

- **If END is encountered in a macro body**
  The end of the source program is assumed and END processing is performed.

(14) For details of the relationship between MACRO and INCLUDE, see **(1)** in **Section 19.1.1**.

[Example]

[Example 1] Macro having no parameters

```
LABEL              MNEMONIC      OPERAND        COMMENT
ADMAC              MACRO                        ;Macro definition
                   MOV           MEM0, #2H
                   ADD           R0, MEM0
ENDM
                     ⋮
                   ADMAC                        ;Macro reference
Macro expansion      ↓

                   MOV           MEM0, #2H
                   ADD           R0, MEM0
```

[Example 2] Macro having a parameter

The following macro subtracts immediate data SB2 from immediate data SB1, then stores the result at the address indicated by data memory MEM0.

```
LABEL              MNEMONIC      OPERAND        COMMENT
SUBMAC             MACRO         SB1,SB2
                   MOV           MEM0, #SB1     ;;SB1 – SB2
                   SUB           MEM0, #SB2
                   ENDM
```

A macro for which parameters are described in the operand field can replace the data at the positions at which parameters are described with any data when the macro is referenced. To replace the values of SB1 and SB2 in the above example, specify the following:

```
LABEL              MNEMONIC      OPERAND        COMMENT
                   SUBMAC        01H,02H
Macro expansion      ↓

                   MOV           MEM0, #01H
                   SUB           MEM0, #02H
```

Placing two semicolons before a comment in the macro, as shown in the above example, specifies that the comment (comment in macro definition) is not to be expanded when the macro is expanded.

16.2 REFERENCING A MACRO

[Format]

```
[<label>:][Δ]macro-name[Δ<actual-parameter-list>][Δ][;<comment>]
```

[Function]

References the macro body defined by the MACRO and ENDM statements.

[Notes]

(1) macro-name must be the "macro name" described in the label field of a MACRO and must be defined before the macro is referenced. If an attempt is made to reference a macro that has not yet been defined, an error (F037: Syntax error) occurs. That line becomes invalid.

(2) The following six types of actual parameters can be described. The actual parameters are evaluated after the macro has been expanded.

<1> Numeric constants
<2> Character constants (ASCII or shift JIS characters enclosed in quotation marks)
<3> Symbols
<4> Expressions
<5> Blanks (no description, comma only)
<6> Any character string containing no blanks

(3) The formal parameters are replaced by the actual parameters in the sequence in which they were originally described, starting from the left.

(4) If the number of formal parameters specified upon defining the macro does not coincide with the number of actual parameters specified when the macro is referenced, the following error may occur:

- **If number of actual parameters > number of formal parameters**
  An error (F036: Operand count error) occurs. That line becomes invalid and the macro is not expanded.

- **If number of actual parameters < number of formal parameters**
  The macro is expanded. NULL strings are passed to the remaining formal parameters. NULL strings cannot be evaluated in the operand field. If, however, a value other than 0 is set in common assemble-time variable ZZZLSARG, an error (F036: Operand count error) occurs. That line becomes invalid and the macro is not expanded.

(5) When describing a delimiter (blank, comma, or quotation mark) as an actual parameter, it must be treated as a character constant and be enclosed in a pair of single quotation marks (for details, see **Section 16.5**).

(6) A macro can be referenced in another macro by describing the other macro name in the macro body. In this case, macros can be nested up to 40 levels deep, including the macro expansion levels and repetitive pseudo instruction block levels. If the nesting level reaches 41, an error (A035: Nesting overflow) occurs, and the assembly processing is terminated.

(7) If a macro name is described as a label, the operand of a pseudo instruction, or an instruction, an error (F037: Syntax error) occurs.

[Example]

[Example 1]

As an example, the macro (SUBMAC) defined in **[Example 2]** of **[Example]** of **Section 16.1.1** is referenced. Actual parameters 01H and 02H are substituted into formal parameters SB1 and SB2, respectively.

```
LOC.  OBJ.  M  I   STATEMENT
                          SUBMAC  01H, 02H
                    ┌──────────────────────┐
                    │ MOV     MEM0, #01H   │
                    │ SUB     MEM0, #02H   │
                    └──────────────────────┘
```

Formal parameters SB1 and SB2 are replaced by actual parameters 01H and 02H. The comment (comment in macro definition) statement, described when the macro was defined, following the two semicolons (;;) is not expanded.

[Example 2]

The following is an example of referencing another macro within a macro. In this case, because an IF ... ENDIF block is used in addition to the macros, the nesting level is two.

```
LOC.  OBJ.  M  I   STATEMENT
                   MAC1    MACRO
                           SKLT    MEM0 , #1H
                   ENDM
                   ;
                   MAC2    MACRO
                           SKNE    MEM0 , #1H
                   ENDM
                   ;
                   MAC3    MACRO   EX1
                           IF      EX1
                                   MAC1
                           ELSE
                                   MAC2
                           ENDIF
                   ENDM
                      ⋮
                           MAC3    0FH
```

Expanded as shown below

```
1                  IF      0FH
2                          MAC1
2                          SKLT    MEM0 , #1H
1                  ELSE
1                          MAC2
1                  ENDIF
```

16.3 EXPANDING A MACRO

RA17K analyzes macros by means of the following procedure:

[Function]

(1) A macro body, enclosed between MACRO and ENDM, is stored into the macro table. If two contiguous semicolons (;;) are detected in the definition statement, the characters between ;; and the next carriage return are assumed to be a comment and are ignored (macro registration).

(2) If there is a macro reference after a macro definition within the same module, the corresponding macro body is expanded at the position where the macro name is encountered.

(3) Formal parameters are replaced by actual parameters when the macro is expanded. At this time, only the character strings described in the macro body are replaced. When a macro within a macro is expanded, therefore, the formal parameters of the inner macro are not replaced by the actual parameters of the outer macro. Similarly, the contents of the include file described in the macro by using INCLUDE are not replaced.

(4) The expanded macro body is assembled.

[Note]

(1) Symbols specified as the operands of pseudo instructions or built-in macros cannot be forward-referenced.

[Example]

[Example 1]

```
LOC.  OBJ.  M  I   STATEMENT

                 LABEL       MNEMONIC        OPERAND        COMMENT

                 ADMAC       MACRO           DATA
                             MOV             R0 , #DATA     ;;R0=0.00H Note
                             MOV             R1 , #DATA+1   ;;R1=0.01H Note
                 ENDM
                                  ⋮
                STOP_COND    DAT             0100B
                             ADMAC           STOP_COND
          +                  MOV             R0, #STOP_COND
          +                  MOV             R1, #STOP_COND+1
```

Note When the macro is expanded, a comment prefixed by two semicolons is not expanded.

[Example 2]

RA17K does not support the definition of a macro in another macro. If a macro is defined in another macro, an error occurs, as illustrated in the example below.

```
LOC.  OBJ.  M  I   STATEMENT

                    MAC1      MACRO
                              NOP
                              MAC2     MACRO
                                       MOV     RG1,#2H
                                       RORC    RG1
                                       SET1    RG_F
                              ENDM
                              IF       P=0
                                     MAC2                ;Error (F037: Syntax error)
                              ENDIF
                    ENDM                                  ;Error (F043: Invalid ENDM
                                                          ;statement)
                     ⋮
                              MAC1                        ;Error (F037: Syntax error)
```

Error (F183: Invalid MACRO place) occurs because MAC2 is defined in the definition block of MAC1. Therefore, all statements up to ENDM of MAC2 are skipped, and registered as MAC1. Consequently, MAC2 is not registered, MAC1 being registered with illegal contents.

16.4 SCOPE OF SYMBOLS IN A MACRO

[Outline]

Two types of symbols, global and local, can be specified in a macro.

(1) Local symbols

A symbol defined in a macro that is valid only in the macro that defines that symbol (local symbol). Even, therefore, if the same symbol is re-defined outside the macro, or if the same macro is referenced more than once or a symbol definition statement is created more than once, the "Symbol multi defined" error does not occur.

(2) Global symbols

In some cases, it is necessary to reference a symbol defined in a macro from a point outside the macro. At this time, the symbol is declared to be GLOBAL, allowing it to be referenced from any point within the same module (global symbol).

Note, however, that if a symbol defined by a pseudo instruction other than SET references a macro declared to be GLOBAL more than once, or if that symbol definition statement is created more than once, the "Symbol multi definition" error occurs.

If the same symbol as that defined by the SET pseudo instruction outside a macro is defined within a macro, that symbol is regarded as being a completely different local symbol in the macro, independent of the symbol having the same name outside the macro.

To assign a value to a symbol outside a macro from within a macro, GLOBAL declaration is required.

[Example] To reference global symbols (DATA_A, DATA_B)

[Example 1]

```
LOC.  OBJ.  M  I   STATEMENT
                     LABEL     MNEMONIC     OPERAND          COMMENT

                     MOS       MACRO                         ;;Macro definition
                               GLOBAL       DATA_A,DATA_B
                     LOOP:
                               MOV          MEM00,#DATA_A
                               MOV          MEM01,#DATA_B
                               :
                     DATA_A    SET          DATA_A + 1
                     DATA_B    SET          DATA_B + 1
                               SKT1         FLG_S
                               BR           LOOP
                     ENDM
```

First macro expansion

```
                     DATA_A    SET          00H
                     DATA_B    SET          00H
                               :
                               MOS                           ;Macro reference
               +               GLOBAL       DATA_A,DATA_B
               +     LOOP:
               +               MOV          MEM00,#DATA_A    <- 00H is referenced for
                                                                  DATA_A
               +               MOV          MEM01,#DATA_B    <- 00H is referenced for
                                                                  DATA_B
               +               :
               +     DATA_A    SET          DATA_A + 1       <- DATA_A becomes 01H
               +     DATA_B    SET          DATA_B + 1       <- DATA_B becomes 01H
                               SKT1         FLG_S
               +               BR           LOOP
```

Because symbols DATA_A and DATA_B, defined in the macro, are declared to be GLOBAL, DATA_A and DATA_B are first assigned the values set as a result of the first macro reference at the second and subsequent macro reference.

```
                    MOS                                  ;Macro reference
+                   GLOBAL      DATAA,DATA_B
+       LOOP:
+                   MOV         MEM00,#DATA_A   <- 01H is referenced for
                                                   DATA_A
+                   MOV         MEM01,#DATA_B   <- 01H is referenced for
                                                   DATA_B
+                    ⋮
+       DATA_A      SET         DATA_A + 1      <- DATA_A becomes 02H
+       DATA_B      SET         DATA_B + 1      <- DATA_B becomes 02H
                    SKT1        FLG_S
+                   BR          LOOP

;Reference outside macro
        MOV             MEM00,#DATA_A           <- 02H is referenced for
                                                   DATA_A
        MOV             MEM01,#DATA_B           <- 02H is referenced for
                                                   DATA_B
```

[Example 2] To reference a reference to a local symbol (AA) with a pseudo instruction at the same level

In this case, only backward reference is performed because a local symbol exists at the same level. If a symbol has not been defined, an error (F058: Undefined symbol) occurs.

```
 LABEL     MNEMONIC     OPERAND      COMMENT

 MAC1      MACRO
 AA        SET          1            ;AA=1
           MAC2
           IF           AA           ;AA=1
           ENDIF
 ENDM

 MAC2      MACRO
 AA        SET          2            ;AA=2
           IF           AA           ;AA=2
           ENDIF
 ENDM
```

[Example 3] To reference a local symbol (AA) with a low-level pseudo instruction

In this case, a high-level symbol is referenced because there is no local symbol at the same level. If a high-level symbol has not been defined, an error (F058: Undefined symbol) occurs.

```
LABEL     MNEMONIC     OPERAND     COMMENT

MAC1      MACRO                    ;High-level macro
AA        SET          1           ;AA=1
          MAC2
          IF           AA          ;AA=1
          ENDIF
ENDM

MAC2      MACRO                    ;Low-level macro
          IF           AA          ;AA=1
          ENDIF
ENDM
```

[Example 4] To reference local symbols (AA, BB) with an instruction at the same level

In this case, because a local symbol exists at the same level, that symbol is referenced. However, to forward-reference the symbol defined by the SET pseudo instruction, the most-recently defined value is used as the value of the symbol.

```
LABEL     MNEMONIC     OPERAND       COMMENT

MAC1      MACRO
AA        SET          1             ;AA=1
          MAC2
          MOV          MEM1 , #AA    ;AA=1
          MOV          MEM1 , #BB    ;BB=2   Value set most recently
AA        SET          AA+1          ;AA=2
BB        SET          1             ;BB=1
BB        SET          BB+1          ;BB=2
ENDM

MAC2      MACRO
AA        SET          3             ;AA=3
          MOV          MEM1 , #AA    ;AA=3
          MOV          MEM1 , #BB    ;BB=4   Value set most recently
AA        SET          AA+1          ;AA=4
BB        SET          3             ;BB=3
BB        SET          BB+1          ;BB=4
ENDM
```

[Example 5] To reference a reference to local symbols (AA, BB) with a low-level instruction
In this case, because there is no local symbol at the same level, a high-level symbol is referenced. To forward-reference the symbol defined by the SET pseudo instruction, however, the most-recently defined value is used as the value of the symbol.

```
 LABEL      MNEMONIC      OPERAND        COMMENT

 MAC1       MACRO
 AA         SET           1              ;AA=1
            MAC2
            MOV           MEM1 , #AA     ;AA=1
            MOV           MEM1 , #BB     ;BB=2   Value set most recently
 BB         SET           1              ;BB=1
 BB         SET           BB+1           ;BB=2
 ENDM

 MAC2       MACRO
            MOV           MEM1 , #AA     ;AA=1
            MOV           MEM1 , #BB     ;BB=2   Value set most recently
 ENDM
```

[Scope of symbol in macro]
The reference range of the symbol used in a macro is classified as follows:

(1) Symbol reference sequence of instruction
The symbol reference sequence of an instruction is as follows:

Reference sequence:
<1> Backward reference at same level
<2> Forward reference at same level
<3> Backward reference at higher level
<4> Forward reference at higher level
<5> Backward reference outside macro
<6> Forward reference outside macro

Examples of referencing symbols with the operands of instructions (including DW/DB) are given below.

[Example 1] Reference sequence for symbol AA

```
LOC.    OBJ.    M  I    SOURCE STATEMENT

                        AA     DAT       1                <5>

                1       MAC1
                1       AA     DAT       1                <3>
                2               MAC2
                2               AA        DAT     1        <1>
                2               MOV       MEM1 ,  #AA
                2               AA        DAT     1        <2>
                1       AA     DAT       1                <4>

                        AA     DAT       1                <6>
```

[Example 2] Reference sequence for labels LAB1, LAB2, LAB3, and LAB4

```
        LOC.    OBJ.    M  I    SOURCE STATEMENT

                        1       MAC01
        0002    074F0   1       LAB1:       NOP
                        2                   MAC02
                                  .
                                  .
                                  .
                0C00F   2                         BR    LAB1
                                                                <1> <2>
        000F            2                         LAB1:
                                                  LAB2:
                                            (ENDM)

                0C002   1                   BR  LAB1
                074F0                       BR  LAB2  <3>; Undefined symbol
                                  (ENDM)
                     1            MAC11
                     2                      MAC12
                0C014   2                         BR    LAB3    <4>
                0C015   2                         BR    LAB4
                                            (ENDM)              <5>
0014    074F0   1               LAB3:       NOP
                                  (ENDM)
0015                            LAB4:
0016                            LAB3:
```

[Explanation] Symbols LAB1, LAB2, and LAB3, used in the above example, are handled as follows:

<1> Reference LAB1, defined at the same level (first macro level).

<2> Reference LAB1, defined at the same level (second macro level).

<3> LAB2, defined at the lower macro level, cannot be referenced.

<4><5> Symbols defined at other than the same level sequentially reference the higher levels.

(2) Symbol reference sequence for pseudo instruction

The symbol reference sequence for a pseudo instruction is as follows:

Reference sequence:

<1> Backward reference at same level
<2> Backward reference at higher level
<3> Backward reference outside macro

Examples of referencing a symbol with the operand of a pseudo instruction (except DW/DB) are given below.

[Example 1] Reference sequence for symbol AA

```
LOC.   OBJ.   M  I    SOURCE STATEMENT

                      AA     DAT       1
                                                          <3>
               1      MAC1
               1      AA     DAT       1
                                                     <2>
               2             MAC2
               2             AA        DAT     1
                                                <1>
               2             IF        AA
               2             ENDIF
               2             AA        DAT     1
               1      AA     DAT       1

                      AA     DAT       1
```

[Example 2] Reference sequences for symbols DT1 and DT2

The scope of a symbol referenced with a pseudo instruction (a statement used to make a decision as to whether to generate an address at path 1, such as conditional judgment or a built-in macro instruction).

```
LOC.   OBJ.   M  I    SOURCE STATEMENT

                      DT0    DAT      00H      <1>

              1       MAC01
              1       DT1    DAT      01H      <2>
              2              MAC2

              2              IF         DT0      ; Reference two levels above
               :                        :
              2              ENDIF
              2              IF         DT1      ; Reference one level above
               :                        :
              2              ENDIF
              2              DT1        DAT   00H
               :                        :          <3>
              2              IF         DT1       ; Reference same level
               :                        :
              2              ENDIF
              2              IF         DT2       <4> ; Undefined symbol
               :                        :
              2              ENDIF
              2           ENDM
              1       DT2 DAT      02H

              1           IF       DT1           <5> ; Reference same level
               :                    :                  (Same DT1 as <2>)
              1           ENDIF
```

[Explanation] Because the symbol used for a pseudo instruction must be resolved in path 1, a previously defined symbol must not be used at the same level. If the symbol is not defined in a subsequent statement at the same level, the level immediately above is searched.

Symbols DT1 and DT2, used in the above example, are handled as follows:

- **<1>** Sequentially searches subsequent statements. References DT0 two levels above.
- **<2>** Sequentially searches subsequent statements. References DT1 one level above (DT1, defined prior to the current level, cannot be seen).
- **<3>** Sequentially searches subsequent statements. References DT1 at the same level.
- **<4>** Because the symbol defined one level above cannot be seen, the "Undefined symbol" error occurs.
- **<5>** References DT1 at the same level.

16.5 MACRO PARAMETER

The formal parameters described in the operand field when a macro is defined are replaced by the actual parameters when the macro is expanded.

Examples of macro definition and macro reference using parameters are given below.

[Example 1]

```
 LABEL      MNEMONIC     OPERAND       COMMENT
 PMAC         MACRO      P1,P2,P3                                  <1>
 L1           DAT        1000H
              DW         P1          ;Formal parameter: P1
              DW         P2          ;Formal parameter: P2
              DW         P3          ;Formal parameter: P3
 ENDM
               :
 L1           DAT        2000H                                     <2>
               :
              PMAC        3000H, L1, 'L1'
 L1           DAT        1000H
              DW         3000H       ;Actual parameter: replaced by 3000H  <3>
              DW         L1          ;Actual parameter: replaced by L1     <4>
              DW         L1          ;Actual parameter: replaced by 'L1'
               :
```

<1> P1, P2, and P3 in the macro definition statement are formal parameters and are referenced by the DW instruction in the macro.

<2> When the macro is referenced, formal parameters P1, P2, and P3 are replaced by actual parameters 3000H, L1, and 'L1'.

<3> Because actual parameter 3000H of P1 is a constant, the constant replaces P1 as is.

<4> Actual parameter L1 of P2 is a symbol, and actual parameter 'L1' of P3 is a character constant. Consequently, the operands of both the DWs are replaced by L1.
This L1 is a local symbol, and is assigned a value of 1000H as defined by the DAT instruction in the macro.

[Example 2]

```
 LABEL       MNEMONIC      OPERAND        COMMENT
 PMAC          MACRO         P1
               GLOBAL        L1
 L1            SET           P1             ;Formal parameter: P1
               IF            .DEF.L1=0
               SET1          L1
               ENDIF
                 :
 L2            SET           L1
                 :
 ENDM

               PMAC          FLG12
               GLOBAL        L1
 L1            SET           FLG12          ;Actual parameter: replaced by FLG12
               IF            .DEF.L1=1
               SET1          L1
               ENDIF
                 :
 L2           SET            L1
                 :
```

Because L1 is declared to be GLOBAL, it is valid even outside the macro. Because both L1 and L2 define symbols by using the SET pseudo instruction, the "Symbol multi defined" error does not occur no matter how many times the macro may be referenced.

[Example 3]

Macro parameters can be described in all the symbol, mnemonic, operand, and comment fields.

```
 LABEL       MNEMONIC      OPERAND        COMMENT
 ABC           MACRO         P1,P2,P3,P4
                 :
 P1:           P2            P3,#0H         ;P4
                 :
 ENDM

               ABC           LOOP, SKT, MEM03, CONVINIENCE!!
                 :
 LOOP:         SKT           MEM03, #0H       ;CONVINIENCE!!
                 :
```

If no actual parameter is described upon referencing a macro for which formal parameters are described, the operand or the number of operands is not evaluated and an error occurs. If a value other than 0 is specified for assemble-time variable ZZZLSARG, however, the error does not occur.

[Example]

```
 LABEL      MNEMONIC     OPERAND      COMMENT
 MAC1         MACRO        P1
              IF           P1
                 AND       MEM0, #1H
              ENDIF
 ENDM
                :
              MAC1
```

Because an error (F036: Operand count error) occurs because no actual parameter is defined when MAC1 is referenced, the macro is not expanded.

To reference the single quotation marks as parameters, describe two contiguous quotation marks.

[Example]

```
LOC.  OBJ.  M  I   STATEMENT
                     LABEL    MNEMONIC          OPERAND

                     MAC1     MACRO             A
                              MAC2              A
                     ENDM
                       :
                     MAC2     MACRO             B
                              MAC3              B
                     ENDM
                       :
                     MAC3     MACRO             C
                              C
                     ENDM
                       :
         1                    MAC1              '''''''MOV MEM12, #5H'''''''
         2                    MAC2              '''MOV MEM12, #5H'''
         3                    MAC3              'MOV MEM12, #5H'
   0125  3                         MOV MEM12, #5H
```

In the above MACRO reference, seven quotation marks are described as the operand of MAC1. Of these, the first quotation mark indicates a character constant. The following six quotation marks are three pairs, i.e., three characters. As a result, three quotation marks are passed as parameters to MAC1. Likewise, only one quotation mark is passed to MAC3 in MAC2.

Remark If two contiguous quotation marks are described as the operand of a macro reference statement, and immediately followed by a comma or CR/L_F, a NULL STRING is passed as a parameter. The same applies when the quotation marks are immediately followed by a space or TAB and a comment.

If MAC3 is referenced as follows in the above example, an error occurs.

```
MAC3   '' NOP''
```

In the above example, the first quotation mark is a delimiter indicating a character constant. Note the character following the subsequent quotation mark. If this character were CR/L_F, space, or TAB, a NULL STRING would be passed as a parameter, as discussed above. In this example, however, because an N follows the second quotation mark, an error (F037: Syntax error) occurs.

When nesting macros and repetitive pseudo instructions, a formal parameter must not be described repetitively.

[Example]

```
LABEL    MNEMONIC          OPERAND
ALLOC    MACRO             P1
             IRP           P1,1,2,3
             MEM&&P1&&P1     MEM      0.0&&P1&&P1&&H
             ENDR
ENDM
              :
         ALLOC             0
```

In this example, when formal parameter P1 is replaced by the actual parameter upon expansion of the macro, it is not clear whether P1 is the formal parameter for IRP or ALLOC. In this case, P1 is judged to be the formal parameter for IRP, and is replaced by the actual parameter. In the above example, an error occurs because ALLOC 0 is expanded as follows.

```
MEM&1&1    MEM              0&1&1H        -> Error occurs
```

When nesting a macro in another macro, a formal parameter cannot be assigned to the macro nested at the higher level. The formal parameter is valid only in that macro (the formal parameter is passed as is).

[Example]

```
LABEL       MNEMONIC        OPERAND         COMMENT
MAC1        MACRO               P1
            DW              P1
            ENDM

MAC2        MACRO               P1
            DW              P1
            MAC1            L1
            ENDM
            MAC2            LLL
            DW              LLL
            DW              L1
```

To pass a parameter to the macro in the next-higher level, describe the formal parameter of the macro in the next-lower level as the actual parameter of that macro.

```
LABEL       MNEMONIC        OPERAND         COMMENT
MAC1        MACRO               P1
            DW              P1
            ENDM

MAC2        MACRO               P1
            DW              P1
            MAC1            P1
            ENDM

            MAC2            LLL
            DW              LLL
            DW              LLL
```

16.6 MACRO OPERATORS AND PSEUDO INSTRUCTIONS

RA17K supports special operators and pseudo instructions to make the best use of macros.

16.6.1 Replacement Operator &

[Format]

&

[Function]

If there is a replacement operator (&) in a macro definition statement, the character strings on both sides of & are coupled when the macro is expanded (as a result, the character strings are replaced by a NULL STRING). An & in a character constant, however, is not treated as a replacement operator & and, therefore, the constant is not replaced.

[Notes]

(1) The replacement operator (&) is meaningful only when described in a macro definition statement. If it is described outside a macro definition statement, it causes an error (F037: Syntax error) because it is a special character. The line on which & is described becomes invalid.

(2) An & does not have to be described between a macro formal parameter and a delimiter such as a space, colon (:), semicolon (;), operator, #, or @.

(3) If macros are nested, the formal parameters are sequentially replaced by actual parameters, starting from the macro at the highest nesting level.

(4) The replacement of parameters can be delayed by using two or more replacement operators (&) with a complicated macro having a deep nesting level. Generally, the same number of replacement operators as the number of nesting levels must be used.
For example, a replacement operator having X as a dummy parameter is used two times in the following macro definition, replacement always being performed while the IRP pseudo instruction is executed.

```
LOC.  OBJ.  M  I   STATEMENT
                    ALLOC     MACRO          X
                                 IRP         Z, 1, 2, 3
                                 MEM&&X&&Z       MEM      0.0&&X&&Z&H
                                 ENDR
                    ENDM
                              ALLOC        6
```

First, the formal parameter of IRP, having the deepest nesting level, is replaced. Then, the formal parameter of ALLOC is replaced. At this time, replacement is executed as shown below.

```
MEM&&X&&Z     MEM     0.0&&X&&Z&H
        ↓
MEM&X&&Z      MEM     0.0&&X&&Z&H    <1>
        ↓
MEM&X&Z       MEM     0.0&&X&&Z&H    <2>
        ↓
MEM&X&1       MEM     0.0&&X&&Z&H    <3>
        ↓
MEM&X&1       MEM     0.0&X&1H       <4>
        ↓
MEM61         MEM     0.061H         <5>
```

<1> "MEM" is skipped because it is not a formal parameter. Because "&&" follows, the next character string is checked. Because the next character string is a formal parameter, one "&" is deleted.

<2> Because the next character string "X" is the formal parameter of the macro one level below, it is not replaced by an actual parameter here.
Because "&&" appears again, and because the character string preceding "&&" is a formal parameter, one "&" is deleted.

<3> Checking reveals the next character string to be a formal parameter. This formal parameter is also found for IRP. Therefore, it is replaced by actual parameter "1".

<4> In the same manner as in **<1>** through **<3>** above, the entire line is processed up to the end.

<5> Next, the formal parameter of macro ALLOC one level below is replaced by an actual parameter.

<6> **<1>** through **<5>** are repeated as many times as the number of actual parameters.

The formal parameters are replaced as soon as the macro is called. However, formal parameters X and Z are replaced when the IRP pseudo instruction is expanded. The above example (ALLOC 6) is ultimately expanded as follows.

```
MEM61     MEM     0.061H
MEM62     MEM     0.062H
MEM63     MEM     0.063H
```

(5) To describe replacement operator "&", consider the following:

- **To describe "&" between a character constant and formal parameter**
  Describe "&" as many times as the number of nested macros, from the position where the formal parameter is to be described to the macro for which the formal parameter is defined.

```
ABC     MACRO   P1
    IRP     P2,1,2,3
        IRP     P3,1,2,3
            MEM&P3              Because a formal parameter is used in a defined
                                macro, one "&" is described.
            MEM&&P2             Because a formal parameter defined in a macro one
                                level below is used, "&" is described twice.
            IRP     P4,1,2,3
                MEM&P4
                MEM&&P3
                MEM&&&P2
                MEM&&&&P1
            ENDP
        ENDP
    ENDP
ENDM
```

- **To describe "&" between two formal parameters**
  Describe "&" as many times as the number of times macros are nested, from the position at which one of the two formal parameters for which the macro nesting level is greater.

```
ABC     MACRO   P1
    IRP     P2,1,2,3
        IRP     P3,1,2,3
            IRP     P4,1,2,3
                MEM&P4&&&&P1
                                Describe "&" as many times as the number of
                MEM&&P3&&P4     occurrences of P4 because the macro nesting level
                MEM&&&P2        for P4 is greater than that for P1.
                MEM&&&&P1&&&&P1
            ENDP
        ENDP
    ENDP
ENDM
```

(6) The characters in a character constant cannot be replaced by a formal parameter. Therefore, even if "&" is described as a character in a character constant, "&" does not disappear when the macro is expanded.

```
ERRORGEN    MACRO       A, B
            ZZZERROR    'A&B ERROR ILLEGAL STATEMENT'     ;

ENDM

            ERRORGEN    PAGE,100
            ZZZERROR    'A&B ERROR ILLEGAL STATEMENT'
```

As a result of expanding this macro, the message 'A&B ERROR ILLEGAL STATEMENT' is output, and parameters A and B are not replaced by '100' and 'PAGE'.

(7) Because formal parameters can be described in the comment field, the replacement operator can also be described in the comment field.

16.6.2 Comment in Macro Definition

[Format]

```
;;<comment>
```

[Function]

The comment in a macro definition is an arbitrary test (comment) described in the macro definition. <comment> prefixed by two semicolons is ignored and not output when the macro is expanded. On the assembly listing, the comment is output only on the macro definition line.

Conversely, if a normal comment statement that is described immediately after one semicolon is described in a macro definition, that comment is output even when the macro is expanded.

Therefore, a comment in a macro definition is used as a comment that need not be output when the macro is expanded.

16.6.3 Expression Operator %

[Format]

```
%<symbol>
%(<expression>)
```

[Function]

If expression operator (%) appears in a macro, the <symbol> or <expression> immediately following % is replaced by a 32-bit numeric value. This value is always prefixed with 0 and suffixed with H.
To couple expression operator % <expression> with other character strings, describe % <expression> and the character strings in succession, in the same way as when coupling using &. Therefore, <character-string>+%<expression> in a macro definition block is replaced by <character-string>+<evaluation-result-of-expression> when the macro is expanded.

[Example]

Definition		Expansion
CNT SET 1		
M%CNT	->	M01H
M% (CNT) N	->	M01HN
M% (CNT+1) N	->	M02HN
M%CNT+1	->	M01H+1

[Notes]

(1) The expression operator (%) can be described only in a macro definition statement. If it is described in any other statement, it causes an error (F037: Syntax error) because it is a special character. The line on which % is described becomes invalid.

(2) If a reserved word (except functions having no argument and assemble-time variables) is described as the symbol of %<symbol>, an error (F037: Syntax error) occurs. The line on which the reserved word is described becomes invalid.

(3) If an undefined symbol, forward reference symbol, or external definition symbol is described as the symbol of %<symbol>, an error (F058: Undefined symbol) occurs. That line becomes invalid.

(4) %<expression> can be used only as a macro parameter or within a macro. If it is used outside a macro, an error (F037: Syntax error) occurs. That line becomes invalid.

(5) The symbol described in <expression> must be defined before macro reference. If a symbol defined after the macro reference line or an undefined symbol is described, an error (F058: Undefined symbol) occurs. That line becomes invalid.

(6) If the description of <expression> is invalid, an error occurs (for details, see **Section 4.7.1**). That line becomes invalid.

(7) Because % is replaced by a NULL STRING after <expression> has been evaluated, & need not be described. However, even if & is described as shown in the example below, an error does not occur.

[Example]

```
M     LABEL     MNEMONIC     OPERAND          COMMENT

      CNT       SET          5

      M1        MACRO
                  ADD        MEM&%CNT , #1
      ENDM

                M1
+                 ADD        MEM05H , #1
```

(8) To describe %<expression> in a macro definition block, macro parameters may be used in <expression>.

[Example]

```
M     LABEL     MNEMONIC     OPERAND          COMMENT

      M1          MACRO      p1
                    ADD      MEM%p1 , #1
      ENDM

      CNT         SET        5
      M1          CNT
+                   ADD      MEM05H , #1
```

In the above example, formal parameter p1 is replaced by actual parameter CNT, then converted into a character string by %.
The following description can also be made.

```
M     LABEL     MNEMONIC     OPERAND          COMMENT

      M1          MACRO      p1
                    ADD      MEM&p1 , #1
      ENDM

      CNT       SET          1
      M1        %CNT
+                   ADD      MEM01H , #1
```

(9) Only a DAT type expression can be described as <expression>. If any other type of expression is described, an error (F045: Invalid type) occurs. That line becomes invalid.

[Example]

[Example 1] To pass as a macro parameter

```
LOC.  OBJ.  M  I   STATEMENT
                    LABEL     MNEMONIC          OPERAND      COMMENT

                    VAL_DEF   MACRO             SUFFIX
                              GLOBAL            M&SUFFIX
                              REPT              10
                                   M&SUFFIX     MEM          M00H + SUFFIX
                                   A            SET          A + 1
                              ENDR
                    ENDM
                    ;
                    A         SET               1
                    M00H      MEM               0.00H
                    ;
                              VAL_DEF      %A
```

The following statements are created by expanding this macro.

```
                    M01H      MEM       M00H + 01H
                                ⋮
                    M0AH      MEM       M00H + 0AH
```

[Example 2] To describe in a macro (example of structured macro)

```
LOC.  OBJ.  M  I   STATEMENT

                    OPJDGE        MACRO       p4
                            GLOBAL    OPERATOR
                            IFCHAR       p4, =
                              OPERATOR           SET     1
                            ENDIFC
                            IFCHAR       p4, >=
                              OPERATOR           SET     2
                            ENDIFC
                            IFCHAR       p4, <<
                               OPERATOR          SET     3
                            ENDIFC
                            IFCHAR       p4, !=
                               OPERATOR          SET     4
                            ENDIFC
                    ENDM

                    IFX           MACRO       p1,p2,p3
                            GLOBAL   L_CNT
                            GLOBAL   B_CNT
                            IF         NOT.DEF.L_CNT
                              L_CNT    SET        0
                            ENDIF
                            IF         NOT.DEF.B_CNT
                              B_CNT    SET        0
                            ENDIF
                    ;
                            L_CNT    SET     L_CNT + 1
                            B_CNT    SET     B_CNT + 1
                    ;
                            OPJDGE           p2
                    ;
                            CASE             OPERATOR
                              1:
                                    SKE      p1, #p3
                                    BR       LAB%(L_CNT)_%(B_CNT + 1)
                              2:
                                    SKGE      p1, #p3
                                    BR        LAB%(L_CNT)_%(B_CNT + 1)
                              3:
                                    SKLT      p1, #p3
                                    BR        LAB%(L_CNT)_%(B_CNT + 1)
                              4:
                                    SKNE      p1, #p3
                                    BR        LAB%(L_CNT)_%(B_CNT + 1)
                            ENDCASE
                    ENDM
```

```
ELSEX          MACRO
        GLOBAL   B_CNT
        B_CNT    SET     B_CNT + 1
        ;
                 BR       LAB%(L_CNT)_%(B_CNT + 1)
        LAB%(L_CNT)_%(B_CNT):
ENDM

ENDIFX          MACRO
        GLOBAL   B_CNT
        B_CNT    SET     B_CNT + 1
        ;
        LAB%(L_CNT)_%(B_CNT):
                 BR       LAB%(L_CNT)_%(B_CNT + 1)
        ;
        GLOBAL  L_CNT
        L_CNT   SET     L_CNT – 1
ENDM
```

Remark IFX, ELSEX, and ENDIFX are used as follows.

```
IFX     MEM010, != , 3
  LOOP:
        SET1    IPTM
        SKF1    IRQTM
        BR      INT_TM_TABLE
        BR      LOOP

ELSEX
        NOP

ENDIFX
```

```
        SKNE    MEM010, #3
        BR      LAB1_2
           LOOP:
                SET1    IPTM
                SKF1    IRQTM
                BR      INT_TM_TABLE
                BR      LOOP

        BR      LAB1_3
     LAB1_2:
                NOP

     LAB1_3:
```

16.6.4 GLOBAL

Symbols defined in a macro are usually local symbols and cannot be referenced from outside the macro.

To reference such symbols from outside the macro, they must be declared as global symbols by using the GLOBAL pseudo instruction before the symbols are defined.

[Format]

```
GLOBAL∆<symbol-list>[∆][;comment]
```

[Function]

(1) Declares that the symbols described in operand <symbol-list> are global symbols that can be referenced from outside the macro.

(2) Allows symbols defined outside the macro by the SET pseudo instruction to be manipulated within the macro by using the SET pseudo instruction.

[Notes]

(1) The GLOBAL pseudo instruction can be described only in a macro definition block. If it is described in any other block, an error (F145: Impossible to use out of macro) occurs. That line becomes invalid.

(2) More than one symbol name can be described as the operand of the GLOBAL pseudo instruction. As many symbols as required can be described, provided the maximum number of characters per line (253) is not exceeded.

(3) If a symbol described as the operand of the GLOBAL pseudo instruction has already been defined at the backward of the same macro level, an error (F057: Symbol multi defined) occurs. That line (global declaration) becomes invalid.

(4) If the symbol defined by the SET pseudo instruction references forward, the value SET last is referenced.

(5) If <label> is described for a GLOBAL statement, an error (F037: Syntax error) occurs, and that one line (global declaration) becomes invalid.

[Example]

[Example 1]

The symbol declared to be GLOBAL in a macro remains valid even after the macro has been expanded.

```
LOC.  OBJ.  M  I  STATEMENT
                  LABEL     MNEMONIC     OPERAND      COMMENT

                  STMAC     MACRO
                            GLOBAL       SYMA
                  SYMA      SET          00H          ;Macro definition
      0000                  DB           SYMA
                  ENDM
                              ⋮
                            STMAC                     ;Macro expansion
                              ⋮
      0000                  DB           SYMA         ;Referenced from
                                                      ;outside of macro
```

[Example 2]

Symbol (FLGA), declared to be GLOBAL in the macro, can be referenced from outside the macro. The value of the local symbol (FLGB) must be re-defined outside the macro.

```
LOC.  OBJ.  M  I  STATEMENT
                  LABEL     MNEMONIC     OPERAND      COMMENT

                  BAIGMAC   MACRO
                            GLOBAL       FLGA
                  FLGA      FLG          0.10H.0
                  FLGB      FLG          0.10H.1
                            SET1         FLGA
                            CLR1         FLGB
                  ENDM

                            BIGMAC                    ;Macro reference
                              ⋮
                  ;References FLGA, FLGB outside macro
                            SKT1         FLGA         ;FLGA=0.10H.0
                            SKF1         FLGB         ;S error (Undefined
                                                      ;symbol)
```

[Example 3]

A symbol defined outside the macro can be used as a separate symbol in the macro. Therefore, a separate value can be defined for that symbol in the macro, like symbol DATA1 in the following example. Once the macro has been expanded, however, the value of the symbol is restored to the original value that was defined outside the macro.

```
LOC.  OBJ.  M  I   STATEMENT
                    LABEL      MNEMONIC      OPERAND        COMMENT

                    DATA1      DAT           0H
                    MEM01      MEM           0.00H
                    ;
                    SMMAC      MACRO
                    DATA1      DAT           01H
                    MEM01      MEM           0.01H
                               MOV           MEM01,#DATA1   ;MEM01=0.01H
                                                           ;DATA1=1H
                    ENDM
                                 :
                                 :
                                 :
                    ;References DATA1.MEM01 outside macro
                               MOV           MEM01,#DATA1   ;MEM01=0.00H
                                                           ;DATA1=0H
```

[Example 4]

Symbols in the macro can be referenced from any point within the macro.

```
    LOC.   OBJ.   M  I  STATEMENT
                         LABEL      MNEMONIC    OPERAND     COMMENT

           0012          MEM12      MEM         0.12H

                         MAC001     MACRO                   ;Macro definition
                                    GLOBAL      CCC
                                    BR          AAA
                         AAA:
                                    BR          AAA
                         CCC:
                                    BR          BBB
                         BBB:
                                    BR          CCC
                         ENDM

    0001   074F0         AAA:       NOP
    0002   0C005                    BR          CCC
                                    MAC001
                  +                 GLOBAL      CCC
    0003   0C004  +                 BR          AAA
                  +      AAA:
    0004   0C004  +                 BR          AAA
                  +      CCC:
    0005   0C006  +                 BR          BBB
                  +      BBB:
    0006   0C005  +                 BR          CCC
    0007   0C001                    BR          AAA         ;Symbol defined outside
                                                            ;macro is valid
S   0008   074F0                    BR          BBB     058 ;Undefined symbol error
```

[Example 5]

If GLOBAL declaration is not made, even if a symbol is defined by the SET pseudo instruction, the value of the symbol is restored to the original value defined outside the macro, once the macro has been expanded.

```
                         LABEL      MNEMONIC    OPERAND     COMMENT
                         SYMB       SET         00H
                         SYMC       SET         01H
                                      :
                                      :
    ┌────────────────────────────────────────────────────────────┐
    │    PMAC           MACRO                                    │
    │                   GLOBAL     SYMB                          │
    │    SYMB           SET        02H          ;SYMB=2H         │
    │    SYMC           SET        03H          ;SYMC=3H         │
    │                   MOV        M , #SYMB    ;SYMB=2H         │
    │                   MOV        M , #SYMC    ;SYMC=3H         │
    │    ENDM                                                    │
    └────────────────────────────────────────────────────────────┘
                                      :
                                      :
                         MOV        M , #SYMB    ;SYMB=2H
                         MOV        M , #SYMC    ;SYMC=1H
```

If SYMB and SYMC, re-defined in the macro, are referenced from outside the macro again, the re-defined value (2) of SYMB, which has been declared to be GLOBAL, becomes valid, and as does the original value (1) of SYMC.

[Example 6] To reference global symbol (nest) ... to perform reference with a pseudo instruction or instruction at the same level

In this case, the pseudo instruction only backward-references the global symbol, while the instruction both backward- and forward-references the global symbol.

```
LABEL       MNEMONIC      OPERAND          COMMENT
MAC1        MACRO
            GLOBAL        AA
            GLOBAL        BB
AA          SET           1                ;AA=1
             ;
            MAC2                           ;References macro
             ;
            IF            AA               ;AA=3
            ENDIF
            MOV           MEM1 , #AA       ;AA=3
            MOV           MEM1 , #BB       ;BB=4   Value set most recently
BB          SET           2                ;BB=2
ENDM

MAC2        MACRO
            GLOBAL        AA
            GLOBAL        BB
AA          SET           3                ;AA=3
            IF            AA               ;AA=3
            ENDIF
            MOV           MEM1 , #AA       ;AA=3
            MOV           MEM1 , #BB       ;BB=4   Value set most recently
BB          SET           3                ;BB=3
BB          SET           BB+1             ;BB=4
ENDM
```

[Example 7] To reference global symbol (nest) ... to perform reference with a pseudo instruction at a high level

In this case, the pseudo instruction only backward-references the global symbol.

```
LABEL       MNEMONIC     OPERAND        COMMENT
MAC1        MACRO
            GLOBAL       AA
AA          SET          1              ;AA=1
             ;
            MAC2                        ;References macro
             ;
            IF           AA             ;AA=2
            ENDIF
ENDM

MAC2        MACRO
            GLOBAL       AA
            IF           AA             ;AA=1
            ENDIF
AA          SET          2              ;AA=2
ENDM
```

[Example 8] To reference global symbol (nest) ... to perform reference with an instruction at high level

In this case, the instruction both backward- and forward-references the global symbol.

```
LABEL       MNEMONIC     OPERAND        COMMENT

MAC1        MACRO
            GLOBAL       AA
            GLOBAL       BB
AA          SET          1              ;AA=1
             ;
            MAC2                        ;References macro
             ;
            MOV          MEM1 , #AA     ;AA=1
            MOV          MEM1 , #BB     ;BB=2   Value set most recently
BB          SET          1              ;BB=1
BB          SET          BB+1           ;BB=2
ENDM

MAC2        MACRO
            GLOBAL       AA
            GLOBAL       BB
            MOV          MEM1 , #AA     ;AA=1
            MOV          MEM1 , #BB     ;BB=2   Value set most recently
ENDM
```

[Example 9] To reference global symbol ... to pass data

```
LABEL       MNEMONIC      OPERAND       COMMENT

MAC1        MACRO
            GLOBAL        AA            ;Declares global symbol
AA          SET           1             ;Defines global symbol AA (AA=1)
ENDM

MAC2        MACRO
            GLOBAL        AA
            MOV           MEM1 , #AA    ;References global symbol AA
ENDM
```

[Example 10] Definition of global symbol (error occurs)

In the following example, an error (Symbol multi defined) occurs in the GLOBAL declaration statement because a symbol is referenced or defined before the GLOBAL declaration.

```
LABEL       MNEMONIC      OPERAND       COMMENT

MAC1        MACRO
AA          SET           0
            GLOBAL        AA            ;Defined before GLOBAL statement
ENDM

MAC2        MACRO
            MOV           MEM1 , #AA
            GLOBAL        AA            ;Referenced before GLOBAL statement
ENDM

MAC3        MACRO
            IF            AA
            ENDIF
            GLOBAL        AA            ;Referenced before GLOBAL statement
ENDM
```

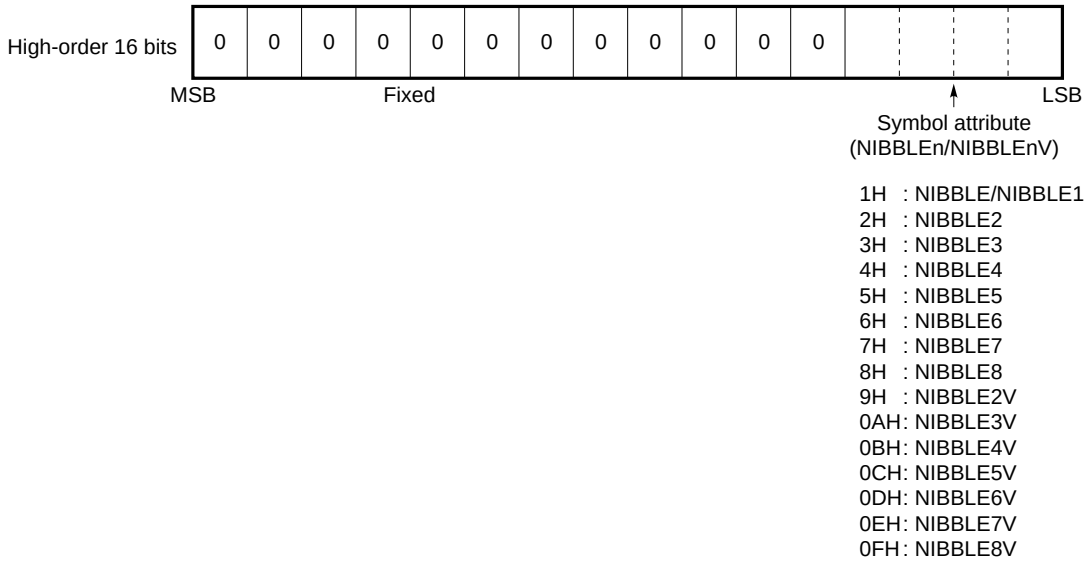
16.6.5 ZZZMCHK

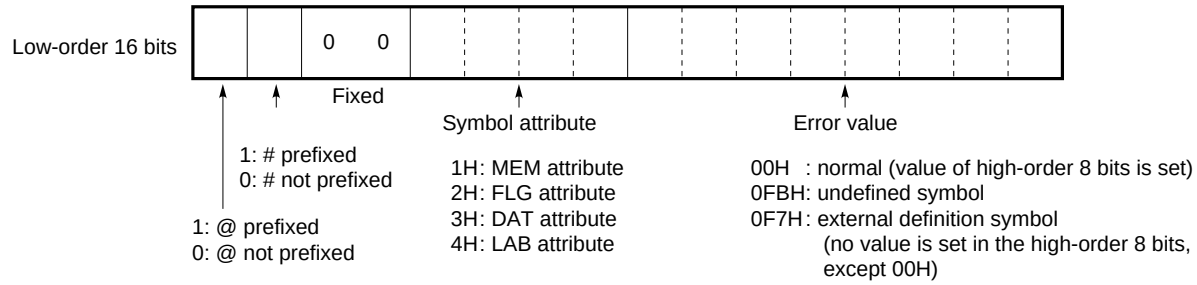
[Format]

<name>ΔZZZMCHKΔ<formal-parameter>

[Function]

Allocates the attribute of the actual parameter symbol replaced as the operand to <name> when the macro is expanded. The value to be assigned to <name> must be of the following format.





If # or @ is prefixed to the symbol name of the operand, # or @ is deleted from the actual parameter after the macro has been referenced by ZZZMCHK.

[Notes]

(1) The information carried by the high-order 16 bits is masked in absolute mode.

(2) ZZZMCHK is meaningful only when it is described in a macro definition statement. If it is described in any other statement, an error (F145: Impossible to use out of macro) occurs. That line becomes invalid.

(3) Data type is allocated to <name>.

(4) If a previously defined symbol is described as <name>, an error (F057: Symbol multi defined) occurs. That line becomes invalid.

(5) If a reserved symbol is described as <name>, an error (F037: Syntax error) occurs. That line becomes invalid.

(6) If <name> is omitted, an error (F037: Syntax error) occurs. That line becomes invalid.

(7) If a character string other than a symbol is described as <formal-parameter>, an error (F070: Invalid operand) occurs. That line becomes invalid. A symbol described as <name> is not registered.

(8) If <formal-parameter> is omitted, an error (F037: Syntax error) occurs. That line becomes invalid.

(9) If an operation expression of MEM type is described as <expression>, nibble information is collected from the operation result.

```
A    NIBBLE2    0.00H
B    SET        .TYPE. (A+A)
```

.TYPE. assigns value 23H to B to collect nibble information from the result of operation A+A.

[Example]

[Example 1]

```
LOC.  OBJ.  M  I   STATEMENT

                    AMAC        MACRO          B,C
                    ;
                    BB          ZZZMCHK        B
                    CC          ZZZMCHK        C
                    ;
                                IF             (BB AND 4F00H) = 4300H
                                   ADD         MEM010, B
                                ELSE
                                   ADD         MEM010, #B
                                ENDIF
                    ;
                                IF             (CC AND 8F00H) = 8300H
                                   BR          C
                                ELSE
                                   BR          @C
                                ENDIF
                    ;
                    ENDM
```

[Example 2] Addition in units of bytes

```
LOC.  OBJ.  M  I   STATEMENT

                ADD2     MACRO            A,B
                ;
                AA       ZZZMCHK          A
                BB       ZZZMCHK          B
                ;
                         IF               (AA AND 0F00H) = 100H    ;MEM TYPE
                            CASE          (BB AND 0F00H)
                            100H:                                  ;MEM TYPE
                                       ADD     A,B
                                       ADDC    A+1,B+1
                                       EXIT
                            300H:                                  ;DAT TYPE
                                       ADD     A, #B AND 0FH
                                       ADDC    A+1,#(B SHR 4) AND 0FH
                                       EXIT
                            OTHER:
                                       ZZZERROR        '2nd OPERAND ERROR'
                            ENDCASE

                         ELSE
                            ZZZERROR         '1st OPERAND ERROR'
                         ENDIF
                ENDM
```

Reference this macro as follows:

<1> Addition of memory by using register

```
ADD2     R1, M1
```

<2> Addition of memory and immediate data

```
ADD2     M1, #N
             ↑
```

In this case, # is deleted by ZZZMCHK in the macro.

16.6.6 PURGE

[Format]

PURGE∆<macro-name>

[Function]

Deletes the macro defined in the macro table reserved in memory or on disk.
When a macro is defined, the macro is registered in the memory of the host machine. If a macro is larger than the macro registration capacity reserved when the assembler was started, the excess portion is automatically written to the disk. Therefore, the absolute capacity of the macro that can be registered depends not only on the reserved memory capacity, but also on the capacity of the disk (disk specified by work drive specification option [/WORK=]) (therefore, the macro can be registered provided the disk capacity permits).
PURGE deletes all unnecessary macros from memory to increase the memory capacity and allow more macros to be registered and thus shorten the assemble processing time (macro reference time).

[Notes]

(1) The PURGE pseudo instruction must not be described in a macro definition statement.
If the PURGE pseudo instruction is encountered during macro expansion, an error (F037: Syntax error) occurs. That line becomes invalid.

(2) If a symbol other than a previously registered macro name is described as the operand of the PURGE instruction, an error (F037: Syntax error) occurs. That line becomes invalid.

(3) If a PURGEd macro name is described in a statement in the source program, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) Even if a PURGEd macro name is defined as a separate forward macro within the same module, the "Symbol multi defined" error does not occur. Moreover, the macro can also be referenced after that macro definition.

(5) If a macro is PURGEd and a vacant macro area becomes available in memory, the macro defined after PURGE is registered in a vacant area in memory.
A macro written to the disk before PURGE remains registered on the disk even if a sufficiently large memory area has become available. In other words, a macro on disk does not automatically move into memory.

(6) If <macro-name> is omitted, an error (F037: Syntax error) occurs. That line becomes invalid.

(7) If a label is described in the PURGE statement, an error (F037: Syntax error) occurs. That line becomes invalid.

[Example]

```
LOC.  OBJ.  M  I   STATEMENT

                    BRANCH   MACRO        ADDR
                             BR           ADDR
                    ENDM
                             BRANCH       1000H      <- Macro expansion
            1                BR           ADDR
                             PURGE        BRANCH
                               :
                             BRANCH       1000H      <- S error
```

[MEMO]

CHAPTER 17 MASK OPTION PSEUDO INSTRUCTION

For details of the mask option pseudo instruction, see the User's Manual for the device file of the target device.

17.1 OPTION ... ENDOP

[Format]

```
OPTION
   ⋮
   ⋮
ENDOP
```

[Function]

The block enclosed between OPTION and ENDOP is used as a mask option definition block.
When the assembler encounters the OPTION pseudo instruction, it automatically includes the mask option definition file (.OPT), registered in the device file.

[Notes]

(1) OPTION must be always terminated with ENDOP. If ENDOP is missing, or if END is encountered before ENDOP, an error (F034: No ENDOP statement) occurs. If an ENDOP instruction is encountered before OPTION, an error (F144: Invalid ENDOP statement) occurs. That line becomes invalid.

(2) If an instruction that creates an object in the OPTION ... ENDOP block is described, a warning message (W068: Operation in OPTION block) is output. At this time, the object for the instruction is not created.

(3) The OPTION ... ENDOP block can be described only once in a group of source programs. If it is described two or more times, the second and subsequent OPTION blocks cause an error (F053: Duplicated OPTION directive) to occur, and all statements preceding ENDOP are ignored. In this case, the data described first is stored to the data storage address (OPDATA) in the ICE file.

(4) The OPTION ... ENDOP block must not straddle two source module files.

(5) Only the object (option data) is output as a list in the OPTION ... ENDOP block.

(6) If a label is described, an error (F037: Syntax error) occurs.

(7) If an error occurs while a macro call statement is being expanded, the error code is not output to the list, instead being output to the log file.

(8) If CSEG is described between OPTION and ENDOP, an error (F172: Invalid CSEG statement) occurs.

(9) If execution terminates by describing OPTION ... ENDOP in REPT and by using EXITR, an error (F034: No ENDOP statement) occurs because ENDOP processing is not performed.

(10) If an error occurs between OPTION and ENDOP, that OPTION pseudo instruction becomes invalid.

(11) The mask option pseudo instruction that actually sets a mask option differs depending on the target device. For details, refer to the user's manual and data sheet for the target device.

CHAPTER 18 CHARACTER STRING REPLACEMENT PSEUDO INSTRUCTIONS

This chapter explains the character string replacement pseudo instructions that are used to replace one character string with another.

The following character string replacement pseudo instructions are supported.

- LITERAL (replaces the subsequent character string with another character string.)
- UNLITERAL (determines the end of the valid range of LITERAL.)

18.1 LITERAL

[Format]

```
LITERAL∆<string-to-be-replaced>,<replacing-string>[∆][;<comment>]
```

[Function]

Replaces <string-to-be-replaced>, described as the first operand, with <replacing-string>, described as the second operand. Once this specification has been made, statements are evaluated by the replaced character string. However, any character strings on the macro expansion line and in comments are not replaced.

The parameters of built-in macro pseudo instructions are replaced. The expansion line, however, is not replaced.

[Notes]

(1) A character string that has been replaced once cannot be replaced again. Therefore, a character string cannot be duplicated, even in the following case.

```
 LABEL      MNEMONIC     OPERAND          COMMENT

 A          MEM          0.01H
            LITERAL      A, B          ; A->B
            ADD          A, #1H        ; Error (F058: Undefined symbol)
             ⋮
            LITERAL      B, A          ; LITERAL B, B
            ADD          B, #1H        ; Error (F058: Undefined symbol)
             ⋮
            ADD          B, #1H        ; Error (F058: Undefined symbol)
```

(2) The character string before replacement is output to the assembly listing, the output object corresponding to the evaluation value of the character string after replacement.
However, the character string of the macro expansion line after replacement is output. This is because the macro body is registered after the character string has been replaced.

(3) This pseudo instruction must not be described in a macro. Otherwise, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) If no more character strings can be registered, an error (A039: Symbol or macro area overflow) occurs. That line becomes invalid, and the character string is not registered.

(5) The macro body is registered with the characters replaced by LITERAL. Consequently, replacement by LITERAL is not performed when the macro is expanded.

(6) When include is performed by executing the INCLUDE pseudo instruction, the character strings of the contents of the included source file are also replaced by LITERAL.
If the INCLUDE pseudo instruction is described in a macro body, however, the instruction is interpreted and include processing is performed when the macro is expanded. Because the macro is being expanded, however, the character strings of the contents of the file included by the INCLUDE pseudo instruction are not replaced by LITERAL.

(7) The operands of LITERAL must not be omitted. If omitted, an error (F037: Syntax error) occurs.

[Example]

[Example 1]
When recycling a program used with the AS17K, character string "LITERAL," used as a symbol, duplicates reserved word "LITERAL" of RA17K. In this case, an error occurs. To prevent this error from occurring, apply the following countermeasures.

```
LABEL     MNEMONIC     OPERAND          COMMENT

          LITERAL      LIT, LITERAL     ; Use 'LIT' instead of reserved
                                        ; word 'LITERAL' (subsequently,
                                        ; 'LIT' is used instead of
                                        ; 'LITERAL').

          LIT          LITERAL, LITERAL1  ; Change existing symbol 'LITERAL'
                                          ; to 'LITERAL1.'
```

[Example 2]
To use a program that uses user-defined symbol POA0 for the target device for which POA0 is defined as a reserved word, in the same manner as in Example 1.

```
BR      POA0
 :                    ] A "Symbol multi defined" error occurs because POA0
 :                      is used as a reserved word.
POA0:
```

In this case, add a character string replacement line at the beginning of the source, and specify a dummy character string at the location where the reserved word is to be used.

```
LITERAL       DUMMY_POA0, POA0
LITERAL       POA0,   LAB_POA0 ---------------------------------+
                                                                 |
BR          POA0                                                 |
 :                    ] Original source                         | Added portion
 :                                                              |
POA0:                                                           |
                                                                 |
SET1        DUMMY_POA0 -------------------------------------------+
```

When this source is assembled, the character string is replaced as follows.

```
LITERAL      DUMMY_POA0, POA0
LITERAL      POA0,   LAB_POA0

BR       POA0                ;(processed with the contents of
                             ;LAB_POA0)
 ⋮
POA0:                        ;(processed with the contents of
                             ;LAB_POA0)

SET1     DUMMY_POA0          ;(processed with the contents of POA0)
```

The original character string is output to the listing. However, the object corresponding to the replaced character string is processed.

[Example 3] Macro expansion after UNLITERAL

```
    LITERAL A, B
    ABC MACRO
        AND A, #1
        ENDM
    UNLITERAL      A

        ABC
```

Character A is replaced by B and registered when the macro body is registered. Consequently, even if the macro is referenced after UNLITERAL, the macro is expanded with the replaced character string. The replaced character string is also output to the intermediate listing.

[Example 4] When replacing a macro name with LITERAL

```
    LITERAL ABC, AAA
    ABC MACRO
        AND A, # 1
        ENDM

    UNLITERAL     ABC

        ABC
```

Macro ABC is replaced by AAA when it is registered. Even if UNLITERAL is subsequently executed, an error (F037: Syntax error) occurs when ABC is described because macro ABC does not exist. If ABC is described before UNLITERAL, that character is replaced by AAA, and the macro is expanded. If AAA is described instead of ABC, the macro is expanded, in the same manner as when ABC is described, because it is registered as AAA.

18.2 UNLITERAL

[Format]

```
UNLITERAL∆<string-to-be-replaced>[∆][;<comment>]
```

[Function]

Specifies the valid range of the LITERAL pseudo instruction. The character strings in the block enclosed by LITERAL and UNLITERAL are replaced by the specified strings.

[Notes]

(1) This instruction is used to specify the valid range of LITERAL. If it is omitted, all character strings subsequent to LITERAL in the module file are replaced with the specified character strings.

(2) <string-to-be-replaced>, described as the operand, must be the same as <string-to-be-replaced> described for LITERAL. If two or more strings are specified by two or more LITERALs, as many UNLITERALs as the number of LITERALs are required to specify the valid range of each LITERAL.

(3) This instruction must not be described in a macro. If described, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) If the character string described as <string-to-be-replaced) is not registered by LITERAL, or if only UNLITERAL is described, an error (F184: No entry characters for LITERAL) occurs. That line becomes invalid.

[Example]

[Example 1]

```
LABEL     MNEMONIC     OPERAND          COMMENT

A         MEM          0.01H
          LITERAL      A, B             ; This block is within the valid
          ADD          A, #1H           ; range of replacement because
          UNLITERAL    A                ; UNLITERAL is described after
                                        ; LITERAL.
          :                             ; A -> B

          ADD          A, #1H           ; Symbol A of this instruction is
                                        ; not replaced.
          :

          LITERAL      B, A             ; The valid range is up to END
          ADD          B, #1H           ; because UNLITERAL is not
          :                             ; described after LITERAL.
          END                           ; B -> A
```

[Example 2]

```
LABEL       MNEMONIC      OPERAND          COMMENT

A           MEM           0.01H
            LITERAL       A,B              ; Because two or more LITERALs are
            LITERAL       C,D              ; described, two or more UNLITERALs
            ADD           A, #1H           ; must be described to specify the
            ADD           C, #1H           ; valid range of the LITERALs.
            UNLITERAL     A
            UNLITERAL     C
```

CHAPTER 19 CONTROL INSTRUCTIONS

This chapter explains the file control instructions, which control the reading and creation of source module files, object files, and listing files by the assembler.

The file control instructions are as follows:

Source input control instructions

- INCLUDE (Includes a source module file.)
- EOF (Indicates the end of an include file.)

Listing output control instructions

- TITLE (Specifies the title of an assembly listing.)
- EJECT (Specifies a line feed in an assembly listing.)
- C14344 (Specifies the output format of the object code in an assembly listing (1-4-3-4-4-bit format).)
- C4444 (Specifies the output format of the object code in an assembly listing (4-4-4-4-bit format).)
- LIST (Cancels NOLIST.)
- NOLIST (Disables statement output to an assembly listing.)

Instructions for controlling false condition block listing output

- SFCOND (Disables the output of false condition blocks to an assembly listing.)
- LFCOND (Cancels SFCOND.)

Instructions for controlling macro expansion listing output

- SMAC and SBMAC (Control the output of the results of macro expansion to an assembly listing (code is output in rows).)
- VMAC and VBMAC (Control the output of the results of macro expansion to an assembly listing (code is output in columns).)
- OMAC and OBMAC (Control the output of the results of macro expansion to an assembly listing (code is output in lines).)
- NOMAC and NOBMAC (Disable the output of the results of macro expansion to an assembly listing.)
- LMAC and LBMAC (Cancel a macro expansion control instruction.)

The assembler (RA17K) does not process control instructions other than the those given above, merely checking the syntax of any other instructions. Instructions other than source input control instructions are processed by the document processor (DOC17K). That is, control by control instructions is not applied to intermediate listing files output by RA17K, these files being output together with control information.

[Notes]

(1) <label> can be written for control instructions other than EOF. If <label> is not written correctly, an error occurs (see **Section 4.3**). Incorrectly specifying <label> invalidates any control instruction written on the same line as <label>.

19.1 SOURCE INPUT CONTROL INSTRUCTIONS

The source input control instructions (INCLUDE and EOF) add the contents of one specified include file to another. The user may specify only constant definitions and macro definitions in an include file. Thus, the source input control instructions can also be used to add items such as common external variables or complex data tables.

19.1.1 INCLUDE

[Format]

```
[<label>:][Δ]INCLUDEΔ'[<path-name>\]<file-name>'[Δ][;<comment>]
```

[Function]

(1) Depending on the include file extension, INCLUDE causes either of two types of processing to be performed, as described later in this section.

(2) If <path-name> is specified, specifying INCLUDE causes the directory to be searched.
For an explanation of INCLUDE with only <file-name> (without <path-name>), see **Section 1.5**.

(3) If a source module file name is specified in <file-name>, an error (F170: Impossible to include a source module file) occurs. That line is invalidated.

[Notes]

(1) If a <file-name> is not specified in <file-name> (e.g., when only a path name is specified, or a file name longer than eight characters is specified), an error (F191: File name error) occurs.

(2) If a file name is specified with extension .DEV, .SEQ, .OPT, or .SYM, an error (F191: File name error) occurs.

(3) An INCLUDE statement can be written in an include file that is itself specified with INCLUDE. This is referred to as the nesting of INCLUDE. The maximum allowable nesting level for macros and the IF statement is 40, while INCLUDE can be nested up to 8 levels deep. If the nesting level is nine or more, an error (F052: Include nesting error) occurs for the line on which the INCLUDE control instruction is written.

(1) When the include file is a source module file

[Function]

The contents of the source module file specified in <file-name> are expanded as if they were specified where <file-name> is placed.

[Notes]

(1) If no extension is specified for <file-name> in the operand field, processing is performed assuming that .ASM has been specified as the extension.

(2) Unless a single quotation mark (') is specified both before and after <file-name>, an error (F037: Syntax error) occurs. The file is not opened.

(3) A file name may include <path-name>. Up to 141 characters can be specified for <path-name> and <file-name> (the maximum supported by DOS).
If 142 or more characters are specified, an error (F037: Syntax error) occurs. The file is not opened.

(4) If the file specified in INCLUDE cannot be found, an error (F061: No include file xxxxxxxx.xxx) occurs.

(5) Statements expanded by an INCLUDE statement are assembled in the same way as those written in the original source program. Thus, the symbols and macros defined in include files can be referenced from outside those include files. In addition, symbols and macros defined outside include files can be referenced from within the include files.

[Relationships between INCLUDE and other pseudo instructions]

[Example 1] Relationship between INCLUDE and MACRO

- **Contents of the source module file**

```
⋮
INCLUDE' ABCD.ASM'
ENDM
⋮
```

- **Contents of include file ABCD.ASM**

```
A   MACRO       X
    IF  X    <>    ZZZBANK
        SETBANK       X
    ENDIF
```

Macro A is registered normally. Although the include file does not contain ENDM, the existence of ENDM immediately after the INCLUDE statement in the source module file causes macro registration to end normally. Macro A is expanded normally when referenced.

[Example 2] Relationship between INCLUDE and a conditional pseudo instruction

- **When an ENDIF statement is written in a source module file**

Contents of the source module file

```
⋮
INCLUDE ' ABCD.ASM'
ENDIF
⋮
```

Contents of include file ABCD.ASM

```
IF  A=B
    BANK2
ELSE
    BANK1
```

Although the include file does not contain ENDIF, the existence of ENDIF in the source module file terminates the IF specified in the include file.
If A=B is true, the line between IF and ELSE is assembled. The lines between ELSE and ENDIF in the source module file are not assembled, however.

- **When only an ELSE statement is specified in the include file**

  Contents of the source module file

```
 ⋮
IF  A=B
INCLUDE ' ABCD.ASM'
ENDIF
 ⋮
```

  Contents of include file ABCD.ASM

```
    BANK2
ELSE
    BANK1
```

  If IF A=B is true, the line between IF and ELSE (BANK2) is assembled. If IF A=B is false, the INCLUDE pseudo instruction is skipped. Thus, the line following the ELSE statement in the include file (BANK1) is not expanded.

(2) When the include file is a symbol definition file (.EQU)

When RA17K is used, it is recommended that symbols be defined in an independent file.
This method of definition enables the following:

<1> Macro optimization using public symbols (Macros cannot be optimized by using PUBLIC...EXTRN.)
<2> High-speed processing by providing the symbol definition file with an independent MAKE function

[Function]

(1) An include file with extension .EQU is processed as a symbol definition file (EQU file).

(2) Upon the termination of the include operation, an intermediate file with extension .SYM (SYM file) is output for the EQU file. The SYM file is created in the same directory as the EQU file.

(3) The time stamps of the EQU and SYM files are compared. If the SYM file has a later time stamp than the EQU file, the EQU file is not reassembled.

[Notes]

(1) If instructions other than symbol definition pseudo instructions exist in an EQU file, an error (F185: Invalid statement in symbol definition file (.EQU)) occurs. Assembly, however, is performed up to the end of the file.

(2) An INCLUDE statement cannot be specified in an EQU file (i.e., include cannot be nested).

(3) If an EQU file is to be included, the corresponding INCLUDE statement must be written before the symbols defined in the EQU file are referenced. If the corresponding INCLUDE statement is written after a symbol is referenced, especially after a symbol is referenced in the operand of a pseudo instruction, an error (F058: Undefined symbol) occurs for the line on which the symbol is referenced.

(4) If an EQU or SYM file is included, the contents of the include file are not output to an intermediate listing file, nor is the line number incremented.

(5) If a symbol is to be referenced in an EQU file, the symbol must be defined within that EQU file. If the symbol is not defined in the EQU file, an error (F058: Undefined symbol) occurs.

(6) If an EQU file is include and an error occurs in the include EQU file, an error (F189: Impossible to create SYM file due to error in EQU file) occurs. The SYM file is not created. If a SYM file having the same name as the EQU file exists, that SYM file is deleted.

19.1.2 EOF

[Format]

```
EOF[Δ][;<comment>]
```

[Function]

EOF indicates the end of the file specified in an INCLUDE statement.

[Notes]

(1) Even if EOF does not appear on the last line of an include file, an error does not occur nor is a warning issued.

(2) If, within a program, EOF is followed by a statement, a warning (W066: Statement after EOF) is issued. No object code is generated for the statements following EOF. Only the line on which EOF is specified and the next line (to output W066) are output to the intermediate list.
For an EQU file, however, the next line is not output because LIST output is not performed. The next line is output to the LOG file and the screen.

(3) If a label or an operand is specified for an EOF statement, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) The EOF control instruction can be specified only in include files. If this control instruction is specified in a file other than an include file (i.e., in a source file), an error (F040: Invalid EOF statement) occurs.

[Example]

In the following example, an include file is used as a macro definition file.
In the example, macro definition statements used commonly in two or more of the modules constituting a source program are stored together in one include file. The definitions in the include file are referenced by including that file.

- **File: MACRO1.ASM**

```
WAIT_OR_K_OFF_SENSE    MACRO   T
                       IF      T <> 0
                          MOVE         4, UTIMER, T
                          CALL         W_OR_KOF
                       ENDIF
ENDM
                       ⋮
                       INCLUDE         'PUBSYM.ASM'
```

- **File: PUBSYM.ASM**

```
                       PUBLIC          W_OF_KOF
                       PUBLIC          K_OF_SENSE
                       ⋮
```

- **Module 1: MOD1.ASM**

```
                INCLUDE          'MACRO1.ASM'
                         ⋮
W_OF_KOF:
                SKF1     KEYJ
                SKF4     KIN3, KIN2, KIN1, KIN0
                BR       KOF_TIME
                RET
                         ⋮
```

In this example, the contents of the include files are expanded in module 1, MOD1.ASM, as follows:

```
STNO   LOC   OBJ  M  I    SOURCE STATEMENT

  20                1                    INCLUDE       'MACRO1.ASM'
 + 1                1      WAIT_OR_K_OFF_SENSE    MACRO   T
 + 2                1                             IF      T <> 0
 + 3                1                                MOVE        4,UTIMER,T
 + 4                1                                CALL        W_OR_KOF
 + 5                1                             ENDIF
 + 6                1      ENDM
 + :                :                              ⋮
 + 10               2                             INCLUDE         'PUBSYM.ASM'
 + 11               2                             PUBLIC          W_OF_KOF
 + 12               2                             PUBLIC          K_OF_SENSE
 + 13               2                              ⋮
 + :                :                              ⋮
                           W_OF_KOF:
                                        SKF1       KEYJ
                                        SKF4       KIN3, KIN2, KIN1, KIN0
                                        BR         KOF_TIME
                                        RET
```

In this assembly listing, the numbers output in column I (field) indicate the include nesting level. The line numbers output in the leftmost column are those assigned in the include file.

Thus, the line number of a symbol output to a cross-reference listing or to a document consists of the line number of the INCLUDE control instruction and that of the file, the two being connected by a hyphen (-).

The following shows an example where line numbers are connected by hyphens for output to a cross-reference listing.

```
RA17K  V1.00 V1 << D17xxx    XREF LIST>>    HH:MM:SS MM/DD/YY PAGE0-001

PROG =

SOURCE = SAMPLE.ASM

SYMBOL              VALUE  CLASS  TYPE  /REF(#DEF)

SEC_SAMPLE0      00000000 Local   SEC /#      0
WAIT_OR_K_OFF_SENSE
                 00000000 Local   MAC /#   20-1,       29
   ⋮                                                                  Line numbers are connected
   ⋮                                                                  by a hyphen.
   ⋮
KOF_TIME         0000000a Local   LAB /#     37,       34
W_OF_KOF         00000002 Public  LAB /#     31,     29-4
```

19.2 LISTING OUTPUT CONTROL INSTRUCTIONS

Listing output control instructions are used with the document processor (DOC17K). RA17K does not support the listing output control instructions, instead outputting an intermediate listing file, the control instructions remaining in the file. The document processor interprets and processes the listing output control instructions remaining in the intermediate listing file output by RA17K.

19.2.1 TITLE

[Format]

```
[<label>:]TITLE'<character-string>'[;<comment>]
```

[Function]

Specifying TITLE causes a form feed in an assembly listing followed by the printing of <character-string>, specified in the operand of the TITLE instruction, on the header line of the next page.

[Notes]

(1) The combined length of the character string specified in TITLE and the TITLE control instruction itself must not exceed the maximum number of characters that can be written on one line (253).

(2) If it encounters a TITLE statement, the document processor causes a form feed and prints the specified <character-string> on the header line of the new page. The document processor does not print single quotation marks. The TITLE statement is printed on the first line of a new page in a listing.

(3) If ' and ' are omitted, an error (F037: Syntax error) occurs. That line becomes invalid.

[Output example]

```
RA17K    V1.0   V1    <<D17005 ASEMBLER LIST>>

PROG = '.....'
                        TEST PROGRAM   <- The title is printed here.
MODULE = ......ASM

E STNO LOC OBJ M I  SOURCE STATEMENT

    1                            TITLE   'TEST PROGRAM'
    2
    3
    4
    :
```

19.2.2 EJECT

[Format]

```
[<label>:][Δ]EJECT[Δ][;<comment>]
```

[Function]

Specifying EJECT causes a form feed in an assembly listing.

[Notes]

(1) The EJECT statement itself is printed before the form feed occurs.

(2) If a mnemonic or an operand is specified for an EJECT statement, an error (F037: Syntax error) occurs. That line becomes invalid.

[Output example]

```
E STNO LOC OBJ M I  SOURCE STATEMENT

    1
    2
    3
    4
    ⋮
```

```
                        EJECT    <- The form feed code is placed on the line immediately
                                    following this line.
```

19.2.3 C14344

[Format]

```
[<label>:][Δ]C14344[Δ][;<comment>]
```

[Function]

C14344 outputs the object code to the object field of an output assembly listing following the C14344 control instruction, in 1-4-3-4-4-bit format. If no output format is specified, C14344 is assumed.
In 1-4-3-4-4-bit format, a 16-bit object code is divided into 1, 4, 3, 4, and 4 bits starting from the MSB, the result being represented in hexadecimal notation. In an instruction for the 17K series, the high-order five bits constitute an operation code. Thus, for a normal instruction, this representation format provides an easier correspondence to the instruction.

[Notes]

(1) Object code is output, in 4-4-4-4-bit format, to an intermediate listing file output by RA17K.

(2) The object code of constant data defined by a DW, DB, or DCP pseudo instruction is output in 4-4-4-4-bit format.

(3) If a mnemonic or an operand is specified for a C14344 statement, an error (F037: Syntax error) occurs. That line becomes invalid.

19.2.4 C4444

[Format]

```
[<label>:][Δ]C4444[Δ][;<comment>]
```

[Function]

C4444 outputs the object code to the object field of an assembly listing following the C4444 control instruction, in 4-4-4-4-bit format. If no output format is specified, C14344 is assumed.
In 4-4-4-4-bit format, a 16-bit object code is divided into 4, 4, 4, and 4 bits starting from the MSB, the result being represented in hexadecimal notation. The 4-4-4-4-bit format is used to output 16-bit data.

[Notes]

(1) The object code of constant data defined by a DW, DB, or DCP pseudo instruction is output in 4-4-4-4-bit format, regardless of the control instruction.

(2) If a mnemonic or an operand is specified for a C4444 statement, an error (F037: Syntax error) occurs. That line becomes invalid.

19.2.5 LIST

[Format]

```
[<label>:][Δ]LIST[Δ][;<comment>]
```

[Function]

LIST specifies the start of the output of an assembly listing that was previously disabled by specifying NOLIST.

[Notes]

(1) The default setting at the start of assembly (at the beginning of the module file) is that an assembly listing is to be output.

(2) If the output of an assembly listing is started by a LIST statement, the LIST statement is not output to the assembly listing.
If, however, a LIST statement is encountered during assembly listing output, the LIST statement is output.

(3) If a mnemonic or an operand is written for a LIST statement, an error (F037: Syntax error) occurs. That line becomes invalid.

[Example]

```
NOLIST
  ⋮            ;An assembly listing is not printed.
LIST
  ⋮            ;An assembly listing is printed.
```

19.2.6 NOLIST

[Format]

```
[<label>:][Δ]NOLIST[Δ][;<comment>]
```

[Function]

NOLIST specifies the stopping of assembly listing output.

[Notes]

(1) A NOLIST statement remains effective until a LIST statement appears or the module file ends. The default setting at the start of module file assembly is that an assembly listing is to be output. Thus, NOLIST must be specified again for another module file.

(2) If the output of an assembly listing is stopped upon a NOLIST statement being encountered, the NOLIST statement is output to the assembly listing.

(3) If a mnemonic or an operand is specified for a NOLIST statement, an error (F037: Syntax error) occurs. That line becomes invalid.

(4) The status of list output control, specified on the line containing NOLIST, is indicated in ZZZPRITNT.

19.3 INSTRUCTIONS FOR CONTROLLING FALSE CONDITION BLOCK LISTING OUTPUT

This section explains control instructions SFCOND and LFCOND, which control listing output for a block that has not yet been assembled because the condition for a conditional assembly pseudo instruction is false.

SFCOND disables listing output of those blocks for which the condition is false, among the condition blocks following SFCOND. In contrast, LFCOND enables listing output for false condition blocks.

SFCOND and LFCOND are invalidated by specifying -COND as an assembly option. Thus, when the assembler is activated, the specification of SFCOND and LFCOND can be changed without having to modify the program.

The instructions for controlling false condition block listing output are effective for the document processor (DOC17K). RA17K does not process the instructions for controlling false condition block listing output, instead outputting an intermediate listing file, the control instructions remaining in the file. The document processor interprets and processes the instructions for controlling false condition block listing output that remain in the intermediate listing file output by RA17K.

19.3.1 SFCOND

[Format]

```
[<label>:][Δ]SFCOND[Δ][;<comment>]
```

[Function]

SFCOND disables the output of an assembly listing for false condition blocks in an IF (IFCHAR, IFNCHAR, or IFSTR) or CASE statement.

[Notes]

(1) The SFCOND statement is effective when the -NOCOND option is specified for the document processor (DOC17K).

(2) At the start of assembly (at the beginning of the module file), processing is performed assuming that LFCOND, described in the next section, is specified as the default.

(3) An SFCOND statement remains effective until an LFCOND statement is encountered or the module file ends.

(4) An SFCOND statement also disables the output of the control instruction written immediately before the portion for which the SFCOND statement disables listing output.
For example, if the output of the IF...ELSE block in an IF...ELSE...ENDIF block is disabled, the output of pseudo instruction IF is disabled, but ELSE is output.

(5) If a mnemonic or an operand is written for an SFCOND statement, an error (F037: Syntax error) occurs. That line becomes invalid.

[Example]

```
                    ⋮                          (The NOCOND option is specified for
                                                assembly)
            SFCOND                      ; Disable listing output for
                                        ; false condition blocks.
COND        SET             0FFH
            IF              COND
                MOV         A , #01H <- Listing output (This line is assembled.)
                MOV         A , #02H <- Listing output (This line is assembled.)
            ELSE
                MOV         A , #0FH <- No listing output (This line is not
                                        assembled.)
                MOV         A , #0EH <- No listing output (This line is not
                                        assembled.)
            ENDIF
                    ⋮
            LFCOND                      ; Enable listing output for
                                        ; false condition blocks.
```

The listing output for the above example is shown below.
Listing output is not performed for the ELSE pseudo instruction itself.

```
E STNO LOC OBJ M I   SOURCE STATEMENT

                    ⋮
            SFCOND

COND        SET             0FFH
            IF              COND
                MOV         A , #01H <- Listing output (This line is assembled.)
                MOV         A , #02H <- Listing output (This line is assembled.)
            ENDIF
                    ⋮
            LFCOND
```

19.3.2 LFCOND

[Format]

[<label>:][∆]LFCOND[∆][;<comment>]

[Function]

LFCOND outputs an assembly listing for false condition blocks.

[Notes]

(1) When the -NOCOND option is specified for the document processor (DOC17K), LFCOND statement cancels the SFCOND (false condition block assembly listing suppression) instruction specified in the source program.

(2) At the start of assembly (at the beginning of the module file), processing is performed assuming that LFCOND is specified as the default.

(3) If a mnemonic or an operand is specified for an LFCOND statement, an error (F037: Syntax error) occurs. That line becomes invalid.

[Example]

```
                  ⋮                                  (The NOCOND option is specified for
                                                     assembly)
          SFCOND                          ; Disable listing output for
                                          ; false condition blocks.
COND      SET             0FFH
          IF              COND
              MOV         A , #01H <- Listing output (This line is assembled.)
              MOV         A , #02H <- Listing output (This line is assembled.)
          ELSE
              MOV         A , #0FH <- No listing output (This line is not
                                      assembled.)
              MOV         A , #0EH <- No listing output (This line is not
                                      assembled.)
          ENDIF
            ⋮
          LFCOND                          ; Enable listing output for
                                          ; false condition blocks.
```

Listing output for the above example is shown below. Listing output is not performed for the ELSE pseudo instruction.

```
E STNO LOC OBJ M I    SOURCE STATEMENT

                        :
                      SFCOND

                COND  SET             0FFH
                      IF              COND
                          MOV         A , #01H <- Listing output (This line is assembled.)
                          MOV         A , #02H <- Listing output (This line is assembled.)
                      ENDIF
                        :
                      LFCOND
```

19.4 INSTRUCTIONS FOR CONTROLLING MACRO EXPANSION LISTING OUTPUT

This section explains the SMAC, VMAC, OMAC, NOMAC, and LMAC control instructions, which are used to control listing output for macro expansion blocks (also covered are the SBMAC, VBMAC, OBMAC, NOBMAC, and LBMAC control instructions for built-in macro instructions).

SMAC and SBMAC perform the same function, as do VMAC and VBMAC, OMAC and OBMAC, NOMAC and NOBMAC, and LMAC and LBMAC. Which of each pair is to be used depends on the type of the macro for which listing output is to be controlled. Use SMAC, VMAC, OMAC, NOMAC, and LMAC for user-defined macros defined in programs. Use SBMAC, VBMAC, OBMAC, NOBMAC, and LBMAC for those macros built into the assembler (RA17K).

SMAC (and SBMAC) disables listing output for the source statements in the macro expansion block specified immediately after it, and performs listing output, in rows, for those object codes generated by executing a call to the macro.

VMAC (and VBMAC) disables listing output for the source statements in the macro expansion block specified immediately after it, and performs listing output, in columns, for those object codes generated by executing a call to the macro.

OMAC (and OBMAC) performs listing output only for those source statements from which object code is generated upon the issue of a macro call, and disables the output of any statements from which object code is not generated.

NOMAC (and NOBMAC) disables listing output for an entire macro expansion block.

LMAC (and LBMAC) enables listing output for an entire expansion block upon the issue of a macro call.

SMAC (and SBMAC), VMAC (and VBMAC), OMAC (and OBMAC), NOMAC (and NOBMAC), and LMAC (and LBMAC) are invalidated by specifying GEN in the [NOGEN/GEN] option. This enables the control specification to be changed without having to modify the program when the assembler is activated.

The instructions used for controlling macro expansion listing output are effective for the document processor (DOC17K). RA17K does not process the instructions for controlling macro expansion listing output, instead outputting an intermediate listing file, the control instructions remaining in the file. The document processor interprets and processes the instructions for controlling macro expansion listing output that remain in the intermediate listing file output by RA17K.

19.4.1 SMAC and SBMAC

[Format]

```
[<label>:][Δ]SMAC[Δ][;<comment>]
[<label>:][Δ]SBMAC[Δ][;<comment>]
```

[Function]

SMAC and SBMAC disable listing output for all statements in the macro and repetitive blocks, instead outputting only object codes.
Object codes are output, in rows, to the assembly listing SOURCE STATEMENT, with up to eight codes per row. If the number of object codes is greater than 8, they are output to multiple rows.

[Notes]

(1) SMAC (and SBMAC) is effective only when -NOGEN is specified as an option for the document processor (DOC17K).

(2) The object codes in an expansion block are output in rows, with up to eight codes per row. In the output, only the location counter value corresponding to the leftmost object is indicated in the LOC. location column of the assembly listing.

(3) When SMAC (or SBMAC) is specified, the macro call statement (or built-in macro instruction) itself is output to the listing.

(4) If a mnemonic or an operand is specified for SMAC (or SBMAC), an error (F037: Syntax error) occurs. That line becomes invalid.

[Example] (Listing output example)

```
LOC.     SOURCE STATEMENT                                    <- Listing header

                                SMAC
          MAC_OUT_CONTROL       MACRO
                                  :
                                ENDM
                                  :
                                MAC_OUT_CONTROL

01240     0C002       0C004       .....           07128   <- Up to eight objects per row
01248     0713F       15780       .....

                                SBMAC
                                  :
                                SET2    FLG00 ,   FLG10   <- Two flags have different
01255     16001       16011                               addresses.
```

19.4.2 VMAC and VBMAC

[Format]

```
[<label>:][Δ]VMAC[Δ][;<comment>]
[<label>:][Δ]VBMAC[Δ][;<comment>]
```

[Function]

VMAC and VBMAC disable listing output for all the statements in macro and repetitive blocks, instead outputting only object code.
Object codes are output, in columns, to the OBJ. assembly listing.

[Notes]

(1) VMAC (and VBMAC) is effective only when -NOGEN is specified as an option for the document processor (DOC17K).

(2) The object codes in an expansion block are output in columns. In the output, the location counter value corresponding to each object code is indicated in the LOC. location column of the assembly listing.

(3) When VMAC (or VBMAC) is specified, the macro call statement (or built-in macro instruction) itself is output to the listing.

(4) If a mnemonic or an operand is specified for VMAC (or VBMAC), an error (F037: Syntax error) occurs. That line becomes invalid.

[Example] (Listing output example)

```
LOC.    OBJ.  M I  SOURCE STATEMENT                          <- Listing header

                                        VMAC
                     MAC_OUT_CONTROL    MACRO                ;User-defined macro
                                         :
                                        ENDM
                                         :
                                        MAC_OUT_CONTROL

01240    0C002 <- Only objects are output, in columns, to the listing.
01241    0C004     (The LOC. column contains the location address corresponding to the object.)
   :       :
   :       :
01248    07128
01249    0713F
01250    15780
           :
           :
                                        VBMAC
                                        SET2    FLG00 ,    FLG10   ;Built-in macro
01255    16001
01256    16011
```

19.4.3 OMAC and OBMAC

[Format]

```
[<label>:][Δ]OMAC[Δ][;<comment>]
[<label>:][Δ]OBMAC[Δ][;<comment>]
```

[Function]

OMAC and OBMAC output to an assembly listing only those statements, existing in the macro and repetitive blocks, from which object code is generated.

[Notes]

(1) OMAC (and OBMAC) is effective only when the -NOGEN option is specified for the document processor (DOC17K).

(2) For a built-in macro instruction, the assembler outputs only those statements from which object code is generated, unless SBMAC or NOBMAC is specified. That is, the processing performed is the same as that performed when LBMAC is specified.

(3) When OMAC (or OBMAC) is specified, the macro call statement (or built-in macro instruction) itself is also output to the listing.

(4) If a mnemonic or an operand is specified for OMAC (or OBMAC), an error (F037: Syntax error) occurs. That line becomes invalid.

[Example] (Listing output example)

```
LOC.    OBJ.  M I  SOURCE STATEMENT                          <- Listing header

                                    OMAC
                    MAC_OUT_CONTROL MACRO
                                    IF      COND
                                        MOV     MEMA , @REGA
                                        ADD     MEMA , #1
                                    ELSE
                                        MOV     MEMA , @REGB
                                        ADD     MEMA , #1
                                    ENDIF
                                     :
                                     :
                                    ENDM
                                     :
                                COND SET     1
                                     MAC_OUT_CONTROL        <- Only subsequent
01240   0C002                        MOV     MEMA , @REGA       statements are output to
01241   0C004                        ADD     MEMA , #1          the assembly listing.
  :       :                           :
01248                                 :
```

19.4.4 NOMAC and NOBMAC

[Format]

```
[<label>:][Δ]NOMAC[Δ][;<comment>]
[<label>:][Δ]NOBMAC[Δ][;<comment>]
```

[Function]

NOMAC and NOBMAC disable listing output for all the statements in macro and repetitive blocks.

[Notes]

(1) NOMAC (and NOBMAC) is effective only when the -NOGEN option is specified for the document processor (DOC17K).

(2) In the listing, only the location counter value corresponding to the first object of the macro appears in the LOC. column of the row immediately following that of the macro call statement or built-in macro instruction.

(3) When NOMAC (or NOBMAC) is specified, the macro call statement (or built-in macro instruction) itself is output to the listing.

(4) If a mnemonic or an operand is specified for NOMAC (or NOBMAC), an error (F037: Syntax error) occurs. That line becomes invalid.

[Example] (Listing output example)

```
LOC.     SOURCE STATEMENT                                    <- Listing header

                                NOMAC
           MAC_OUT_CONTROL      MACRO
                                  :
                                ENDM
                                  :
                                MAC_OUT_CONTROL

 01240

                                NOBMAC
                                  :
                                SET1    FLG00
 01255
                                INITFLG FLG10 , FLG11 , FLG12 , FLG13
001258
```

19.4.5 LMAC and LBMAC

[Format]

```
[<label>:][Δ]LMAC[Δ][;<comment>]
[<label>:][Δ]LBMAC[Δ][;<comment>]
```

[Function]

LMAC and LBMAC enable listing output for all the statements in macro and repetitive blocks.

[Notes]

(1) At the start of assembly (at the beginning of a module file), processing is performed assuming that LMAC (or LBMAC) is specified as the default.

(2) Use LMAC (or LBMAC) to invalidate SMAC (or SBMAC), VMAC (or VBMAC), OMAC (or OBMAC), or NOMAC (or NOBMAC), specified in the source program, when the -NOGEN option is specified for the document processor (DOC17K).

(3) If a statement in a macro contains a listing output control instruction, processing is performed according to the listing output control instruction even though LMAC is specified before the macro is referenced. For example, if SFCOND is specified in a macro as shown in the following example, listing output is not performed for those statements in the false condition block.

```
                                NOMAC
                                :
                MAC_OUT         MACRO
                                SFCOND
                COND            SET         1
                                IF          COND
                                        MOV     MEMA , @REGA
                                        ADD     MEMA , #1
                                ELSE
                                        MOV     MEMA , @REGB
                                        ADD     MEMA , #1
                                ENDIF
                                :
                                ENDM
                                :
                                LMAC    ;Macro MAC_OUT is referenced after
                                        ;LMAC is specified.
                                MAC_OUT
01240  0C002                    MOV         MEMA , @REGA <- Only this and
01241  0C004                    ADD         MEMA , #1       subsequent statements
                                                           are output to the
                                                           assembly listing.
```

(4) If a mnemonic or an operand is specified for an LMAC or LBMAC statement, an error (F037: Syntax error) occurs. That line becomes invalid.

19.5 DOCUMENT CREATION CONTROL INSTRUCTIONS

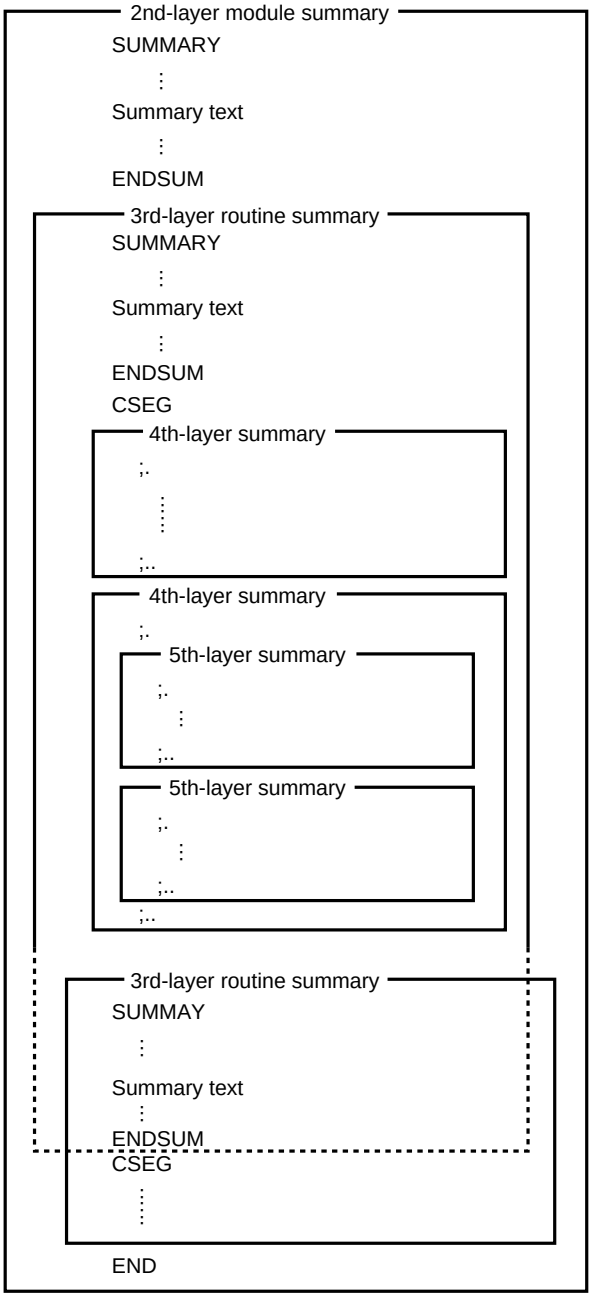
Document creation control instructions are effective for the document processor (DOC17K).

The supported control instructions are SUMMARY and tag (;.). These control instructions are effective only when -SUMMARY is specified as an assembler option. If the -SUMMARY option is not specified, SUMMARY or tag and any subsequent description are assumed to be a comment, such that no processing is performed. In addition, no error occurs nor is any warning issued.

If an error occurs as a result of executing a document creation control instruction, no summary file (.SUM) is created. If only a warning is issued, however, a summary file is created.

[Summary configuration]

The following figure shows the summary configuration in a module (source module file).



Specify a first-layer program summary with the -SUMMARY option of the document processor.

The SUMMARY specified at the beginning of a module is a second-layer module summary.

A SUMMARY that is specified in a location for which a second-layer module summary is not specified becomes a third-layer routine summary.
Normally, specify a third-layer routine summary immediately before CSEG.

Specify fourth- and lower-layer summaries by using a tag (;. and ;..).
The range of a fourth- or lower-layer summary can be specified using ;. and ;... Fourth- and lower-layer summaries can be nested.
Tags can be specified only within section blocks.

19.5.1 SUMMARY

[Format]

Definition format <1>

```
[<label>:][Δ]SUMMARYΔ'<terminating-string>'[,'[\f\nx]<title>[\nx]']

<summary-text> (Character string without a terminating string)
<terminating-string>
```

Definition format <2>

```
[<label>:][Δ]SUMMARY [Δ][,'[\f\nx]<title>[\nx]']

<summary-text>
ENDSUM
```

Remark Parameters \f and \n specify a form feed and a line feed in document text, respectively.

\f : Places the form feed code (FF) immediately before the document text of the module or routine. \f must be specified in half-size lower-case letters. Use the .EJ command to specify a form feed at the end of a document.

\nx : Specifies, with x, the number of line feed codes (CR/L_F) that are to be placed before or after the title to be printed for the document text of the module or routine. Specifying \n immediately before <title> places the line feed code before the title. Specifying \n immediately after <title> places the line feed code after the title. \n must be specified in half-size lower-case letters, while x must be a one-digit decimal number ($1 \leq x \leq 9$).

[Function]

The SUMMARY control instruction defines <summary-text> (summary) of a module or routine.
Two definition formats are supported.

Definition format <1>

The block that begins on the line immediately following that containing the SUMMARY control instruction and which ends with the character string specified in the first operand <terminating-string> is <summary-text> of the module or routine. The second operand is <title> of the module or routine.

Definition format <2>

<terminating-string> can be omitted. When <terminating-string> is omitted, ENDSUM is assumed to be <terminating-string>. In this case, the block between SUMMARY and ENDSUM is assumed to be <summary-text> of the module or routine. The first operand is <title> of the module or routine.

<title> is printed in the contents and text of a document. <summary-text> is printed in the text of a document.
If a NULL character string (' ') is specified for <terminating-string> in definition format **<1>**, only <title> is registered. In such a case, <summary-text> cannot be written.

[Title]

Specify <title> in an operand of the SUMMARY control instruction. <title> can be omitted. When <title> is omitted, it is assigned as described below.

- **When <title> of a module summary is omitted**
  The module file name is <title>.

- **When <title> of a routine summary is omitted**
  The section name is <title>. If no applicable CSEG pseudo instruction is specified or absolute mode is set, <title> is assigned as follows:

  No title xx (Applies to PC-9800 series computers. xx is a serial number consisting of half-size numeric characters, beginning with 01.)
  NON_TITLE xx (Applies to IBM PC series computers. xx is a serial number consisting of half-size numeric characters, beginning with 01.)

[Notes]

(1) <terminating-string> is replaced by a line feed code in the document text. If <terminating-string> is placed at the beginning of a line, one line of line feed codes is placed immediately after the document text.

(2) If a character string containing <terminating-string> is specified in <summary-text>, the portion of the character string that follows <terminating-string> is ignored, such that the definition of <summary-text> ends at <terminating-string>.
If <summary-text> is continued to the next line, an error (F037: Syntax error) occurs.

(3) If a label or an operand is specified for ENDSUM, an error (F037: Syntax error) occurs. That line becomes invalid. This causes the line to be regarded as being part of <summary-text>.

(4) In <terminating-string>, only the first 16 characters (ASCII) are valid. The 17th and subsequent characters, if specified, are ignored.

(5) The combined length of <title> and the SUMMARY control instruction must not exceed the maximum number of characters that can be specified on one line (253).

(6) To specify \ (half-size character) in <title>, code two consecutive \s. For example, to register "no line feed \" as <title>, code the following:

```
SUMMARY          '%','"no line feed \\"'
```

(7) Commands for documents can be specified in <summary-text> (see **[Commands for documents]**, below).
The character string of such a command is not registered as part of <summary-text>.

(8) Be particularly careful when using only one half-size character (8-bit ASCII character) for <terminating-string> in the SUMMARY control instruction of the document creation function.
Assume that one half-size character is specified for <terminating-string>. In this case, if a full-size character specified in the block between SUMMARY and <terminating-string> contains the 8-bit code corresponding to the half-size character specified as <terminating-string>, that full-size character may be regarded as being a terminating character.

[Example] \ is specified as a terminating string.

```
      8-bit code for \             :   5CH
      Shift JIS code for 表 (table) : 955CH

  Example statement:
      SUMMARY       '\','TITLE CHARACTER-STRING'
                     ⋮
                    ･･･表           ;表 is recognized as a terminating
                                     ;string.
                     ⋮              ;All characters following 表 are
           \                         ;handled as syntax errors.
```

(9) The SUMMARY pseudo instruction cannot be nested. Even if a SUMMARY statement is specified between SUMMARY and ENDSUM, that SUMMARY statement is regarded as being part of <summary-text>.

(10) If no <terminating-string> or ENDSUM statement is specified for SUMMARY, an error (F162: No ENDSUM statement) occurs at the end of the END statement or source module file.

(11) Specifying only an ENDSUM statement results in an error (F195: Invalid ENDSUM statement).

(12) If ' ' (NULL character string) is specified in <title>, the assembler assumes that the title has been omitted (see **[Title]**).

(13) If <summary-text> contains an END statement, assembly ends at that END statement.

(14) In the final section, SUMMARY remains effective up to the line preceding the END statement.

(15) SUMMARY remains effective between CSEG and the next CSEG, or between CSEG and the line preceding END.

(16) If multiple SUMMARY statements are specified between CSEGs, a warning (W021: The SUMMARY statement described before is invalid) is issued in the second or a subsequent SUMMARY statement and the most recently written SUMMARY statement becomes valid. That is, the SUMMARY statement specified immediately before CSEG is the routine summary.

[Document commands]

Commands for specifying the document output format are described below. These commands begin with a period (.) and are specified in the block between SUMMARY and a terminating string. The commands can be written at any point within the summary text. A command itself is not output in the document text. RA17K unconditionally skips commands for documents because they are written in summary text. Thus, RA17K does not perform any check on the validity of commands.

<1> .EJECT: In the summary text of document text, places the form feed code at the location where .EJECT is specified.

<2> .EJn: In document text, places the form feed code at the beginning of the n-th line, counted from the line following the end of the summary text.

n = 0 to 250 -> The form feed code is placed on the n-th line. When n = 0, no form feed code is placed. Note that .EJn is invalid in the following cases:

- When the form feed code specified in the ROW option appears before the n-th line.
- When the document text specified in the next SUMMARY statement appears before the n-th line.

n = –1 -> The form feed code is placed at the end of the document text for the module or routine.

n is a decimal number, written in half-size characters, and must be –1 or from 0 to 250. If any other number is specified, an error occurs and the default is assumed.
The default for n is 0, indicating that no form feed code is placed.

<3> .LFn: Specifies the number of line feed codes inserted between titles in the contents.

n = 0 to 99 -> The line feed code is placed on the n-th line, counted from the line immediately following the title. Note that .LFn is invalid if the form feed code specified in the ROW option appears before the n-th line.

n is a decimal number, from 0 to 99, written in half-size characters. If any other number is specified, an error occurs and the default is assumed. The default for n is 0, indicating that no line feed code is placed.

<4> .TITLE'character-string': Specifies the title to be printed on the first line of each page of the document text.

If a page contains documents prepared using multiple SUMMARY statements, the title character string for the document, output on the first line of the page, is printed.

[Notes]

(1) If the .EJn or .LFn command is specified more than once in a single SUMMARY block, the most-recently specified value becomes valid.

(2) Even if the value of n for the .EJn or .LFn command falls outside the specified range, an error does not occur during assembly or linkage. For document processing, however, the document processor (DOC17K) issues a warning (W030: Invalid command). The specified value becomes invalid.

(3) A command must be specified starting from the beginning of a line. It is possible, however, for a command to be preceded by spaces or a tab code.

[Example]

(1) SUMMARY is specified more than once (in absolute mode)

```
SUMMARY ' @'     <-  The module name becomes <title> because <title> is omitted.
  :
<summary-text>
  :
@
SUMMARY ' @'     <-  No title 01 is <title> because <title> is omitted.
  :
<summary-text>
  :
@

SUMMARY ' @'     <-  No title 02 is <title> because <title> is omitted.
  :
<summary-text>
  :
@

END
```

The first SUMMARY specified is the module summary (second layer). Any subsequent SUMMARY is a routine summary (third layer).

(2) SUMMARY is written only once (in absolute mode)

```
SUMMARY ' @' , 'initialization'
  ⋮
<summary-text>
  ⋮
@

END
```

The specified SUMMARY is a routine summary (third layer). The module summary (second layer) is generated automatically by RA17K. The title of the generated module summary (second layer) becomes the module name.

(3) SUMMARY is not specified (in absolute mode)

```
END
```

The module summary (second layer) and a routine summary (third layer) are generated automatically. The title of the generated module summary (second layer) becomes the module name, while the generated routine summary (third layer) has no title.

(4) SUMMARY is specified more than once (in relocatable mode)

```
SUMMARY ' @'     <-  The module name becomes <title> because <title> is omitted.
  :
<summary-text>
  :
@
SUMMARY ' @'     <-  Section name PRO1 is <title> because <title> is omitted.
  :
<summary-text>
  :
@

PRO1     CSEG
             :
             :
             :
SUMMARY ' @'     <-  Section name PRO2 is <title> because <title> is omitted.
  :
<summary-text>
  :
@

PRO2     CSEG
             :
             :
             :
END
```

The first SUMMARY specified is the module summary (second layer). Any subsequent SUMMARY is a routine summary (third layer).

(5) The module summary (second layer) is omitted (in relocatable mode)

```
SUMMARY ' @'     <-  Section name PRO1 becomes <title> because <title> is omitted.
  :
<summary-text>
  :
@

PRO1     CSEG
             :
             :
             :
END
```

If only one SUMMARY is specified between the beginning of a module and CSEG, the SUMMARY is a routine summary (third layer). The module summary (second layer) is generated automatically.

(6) SUMMARY is specified more than once before a CSEG pseudo instruction (in relocatable mode)

```
SUMMARY  ' @'
  ⋮
<summary-text>
  ⋮
@
SUMMARY  ' @'     <-  Although <title> is omitted, no title 01 becomes <title> because no applicable
                      CSEG exists.
  ⋮
<summary-text>
  ⋮
@

SUMMARY  ' @'
  ⋮
<summary-text>
  ⋮
@

PRO1     CSEG
             ⋮
             ⋮
             ⋮
END
```

The first SUMMARY specified is the module summary (second layer). Any subsequent SUMMARY is a routine summary (third layer).

(7) No SUMMARY is specified before a CSEG pseudo instruction (in relocatable mode)
The module summary (second layer) and routine summaries (third layer) are generated automatically.

(8) SUMMARY is specified in CSEG (in relocatable mode)

```
PRO1     CSEG
             :
SUMMARY
 :
<summary-text>
 :
ENDSUM
             :
         END
```

[Relationship between SUMMARY and macros]

If a SUMMARY control instruction is specified in a macro body, it must also be closed within that macro body. If the SUMMARY control instruction is not closed, an error (F162: No ENDSUM statement) occurs once the expansion of all macro bodies is completed. In addition, the SUMMARY control instruction for which the error occurred is closed forcibly upon the occurrence of the error.

(1) Only a SUMMARY statement is specified in a macro body

```
         DAT1     DAT    1
+        MAC1            <- Macro reference
+                 :
+                SUMMARY
+                 :
+        ENDM            <- An error (F162: No ENDSUM statement) occurs when macro
                            expansion is completed, and macro MAC1 is not processed
                            normally. In addition, SUMMARY is closed unconditionally by
                            ENDM, and the next and subsequent lines are assembled.
                 :
         ENDSUM          <- An error (F195: Invalid ENDSUM statement) occurs.
```

19.5.2 ;. (tag)

[Format]

```
[<label>:]  ;.[[\nx]character-string[\nx]]
            ;
            ; <summary-text>
            ;

            statement-group

            ;.. (Tag terminating symbol)
```

[Function]

The character string following ;. is registered as a tag.
The registered character string is used in *SIMPLEHOST* as the lowest-level header of the program layer. In the document, the registered character string is printed in the contents as a low-level header. In the document text, a tag table consisting only of the tags specified in a routine is generated. This tag table is placed after the <summary-text> for the routine.
If \nx is specified immediately before or after character-string, the line feed code is placed either before or after character-string in the contents and tag table. x must be a one-digit decimal number. If \nx is omitted or specified incorrectly, no line feed code is placed.
In relocatable mode, a tag begins with ;. and ends with ;... The scope of a tag can be specified using these symbols. In addition, a tag can be specified within another tag, that is, they can be nested.

[Title]

The title of a tag is determined as follows:

In absolute mode

- The character string following ;. is the title.

In relocatable mode

- **When <label> is specified**
  <label> becomes <title>. The character string following ;. is merely a comment.

- **When no <label> is specified**
  The character string following ;. is <title>.

- **When neither <label> nor the character string following ;. is specified**
  The first line of <summary-text> is the title. If the first line of <summary-text> is a blank line or if <summary-text> is omitted, the title is no title xx (where xx is the serial number).

[Summary text]

In relocatable mode, all comment lines (lines in which only comments are specified) immediately after the line specifying a tag, are <summary-text>. A line without a comment terminates <summary-text>.

[Notes]

(1) The maximum length of a character string corresponds to the maximum number of characters that can be specified on a line.

(2) Specification of the tag control instruction itself can be started from any character position on a line. Even if a tag control instruction and other control instructions are specified on the same line, no error occurs. However, that line will not be processed as a tag.

(3) Tags can be nested in relocatable mode. If, however, the nesting is made incorrectly, a warning (W198: No end mark for tag) is issued in the next CSEG or END statement.

(4) In relocatable mode, a tag starting with (.) and terminating with (..) can be replaced with other characters by using the -TAGSTART and -TAGEND options.

(5) In relocatable mode, a tag can be specified only in a section block. If a tag is specified outside a section block, an error (F200: Invalid tag statement) occurs. That line becomes invalid.

(6) In relocatable mode, a range can be specified using the starting (.) and terminating (..) characters. In absolute mode, the scope of a tag begins with the starting character (.) and ends with the next tag or SUMMARY.

(7) If a tag is specified in relocatable mode, any instruction from which object code is generated must be written in the tag (between . and ..). If an instruction from which object code is generated is placed outside the tag, a warning (W201: No mnemonic to make an object code in a tag) is issued.

(8) If the tag terminating character (..) appears before the specification of a tag starting character (.), an error (F200: Invalid tag statement) occurs.

(9) In absolute mode, the tag terminating character (..) is handled as a comment.

(10) All characters following ;.. are ignored.

(11) The -TAGSTART and -TAGEND options are ignored in absolute mode.

[Example]

(1) Example coding (in absolute mode)

```
          ;. processing-1
      ⋮
   statement-group
      ⋮
          ;. processing-2
      ⋮
```

The scope of the tag for title processing-1 extends to the next tag.

(2) Example coding (in relocatable mode)

```
   ABC       CSEG
   title-1:          ;.
                     ; The character string specified here is <summary-text>.
                     ;
                     ;

               statement-group

                     ;. title-2 (<summary-text> is omitted.)

               statement-group

                     ;..  (Terminating symbol for title-2)
                     ;..  (Terminating symbol for title-1)

               END
```

(3) Specifying tags that cause an error (in relocatable mode)

```
ABC       CSEG
title-1:          ;.
                  ; The character string specified here is <summary-text>.
                  ;
                  ;

            statement-group

                  ;. title-2 (<summary-text> is omitted.)

            statement-group

                  ;..  (Terminating symbol for title-2)

                  ;.
                  ;..
            END           <- A warning (W198:  No end mark for tag) is issued because the
                             nesting of tags is invalid.
```

(4) Object code outside a tag (in relocatable mode)

```
ABC       CSEG
title-1:          ;.
                  ; The character string specified here is <summary-text>.
                  ;
                  ;

            statement-group

                  ;..   <- End of the tag for title title-1

            NOP           <- A warning (W201:  No mnemonic to make an object code in a
                             tag) is issued because this instruction is place outside the tag.
                  ;. title-2 (<summary-text> is omitted.)

            statement-group

                  ;..  (Terminating symbol for title-2)

                  ;.
                  ;..
            END
```

19.5.3 ;.V (registration of labels as tags)

[Format]

```
                    ;.V[\nx]
                    ;
                    ;    <summary-text>
                    ;
<label>:

                    statement-group

                    ;..  (Terminating symbol for tag)
```

[Function]

The label on the line following the line where ;.V[\nx] is specified is registered as a tag in the tag table. Most of the specifications for the tag control instruction also apply to the ;.V control instruction.

[Example]

```
LOC.  OBJ.  M  I   STATEMENT

                    ;.V\n2    ;TAG1
                    TBL1:
                    LIST_ON
                    ;
                    PUBLIC    Delay change
                    EXTRN     LAB:DELAYCHNG
                    EXTRN     MEM:RGO,object-of-control

                    ;.V   ;************************************************
                                          Delay change:              ;***
                          ;************************************************

                        LD     RGO,object-of-control  ;Save object-of-control in RGO.
                        DELAY_CHNG
                        BR       Timer  processing
                         :
                         :
                         :
                        BR       Delay  change
                         :
                         :
                         :
                         :
                    ;     ************************************************
                    ;.                    Timer  processing              ;***
                    ;     ************************************************
                    Timer  processing:

                    ;..
                    ;..
```

In the above example, TBL1, delay change, and timer processing are indicated in the routine tag table as tags of the same routine.

CHAPTER 20 17K SERIES INSTRUCTIONS

This chapter explains the mnemonics and operands used with the 17K series.

20.1 MNEMONICS

[Explanation]

(1) The mnemonics are listed below. Some devices may not support some of the mnemonics shown. For details, refer to the data sheet and user's manual provided with the target device, as well as the user's manual for the device file.

(2) For each mnemonic, the number of operands is fixed. If too many or too few operands are specified for a given mnemonic, an error (F037: Syntax error) occurs and the object code indicating an NOP instruction is generated.

(3) If an invalid operand is specified, an error occurs and the object code indicating an NOP instruction is generated.

The instruction sets are shown below:

Instruction set	Mnemonic	Operand(s)	Operation	Machine code			
				Operation code	Operand(s)		
Addition	ADD	r, m	(r) <- (r) + (m)	00000	m_R	m_C	r
		m, #n4	(m) <- (m) + n4	10000	m_R	m_C	n4
	ADDC	r, m	(r) <- (r) + (m) + CY	00010	m_R	m_C	r
		m, #n4	(m) <- (m) + n4 + CY	10010	m_R	m_C	n4
	INC	AR	AR <- AR + 1	00111	000	1001	0000
		IX	IX <- IX + 1	00111	000	1000	0000
Subtraction	SUB	r, m	(r) <- (r) – (m)	00001	m_R	m_C	r
		m, #n4	(m) <- (m) – n4	10001	m_R	m_C	n4
	SUBC	r, m	(r) <- (r) – (m) – CY	00011	m_R	m_C	r
		m, #n4	(m) <- (m) – n4 – CY	10011	m_R	m_C	n4

Instruction set	Mnemonic	Operand(s)	Operation	Machine code			
				Operation code	Operand(s)		
Logical operation	OR	r, m	(r) <- (r) $\vee$ (m)	00110	m_R	m_C	r
		m, #n4	(m) <- (m) $\vee$ n4	10110	m_R	m_C	n4
	AND	r, m	(r) <- (r) $\wedge$ (m)	00100	m_R	m_C	r
		m, #n4	(m) <- (m) $\wedge$ n4	10100	m_R	m_C	n4
	XOR	r, m	(r) <- (r) $\forall$ (m)	00101	m_R	m_C	r
		m, #n4	(m) <- (m) $\forall$ n4	10101	m_R	m_C	n4
Decision	SKT	m, #n	CMP <- 0, if (m) $\wedge$ n = n, then skip	11110	m_R	m_C	n
	SKF	m, #n	CMP <- 0, if (m) $\wedge$ n = 0, then skip	11111	m_R	m_C	n
Comparison	SKE	m, #n4	(m) – n4, skip if zero	01001	m_R	m_C	n4
	SKNE	m, #n4	(m) – n4, skip if not zero	01011	m_R	m_C	n4
	SKGE	m, #n4	(m) – n4, skip if not borrow	11001	m_R	m_C	n4
	SKLT	m, #n4	(m) – n4, skip if borrow	11011	m_R	m_C	n4
Rotation	RORC	r	-> CY -> $(r)_{b3}$ -> $(r)_{b2}$ -> $(r)_{b1}$ -> $(r)_{b0}$	00111	000	0111	r
Transfer	LD	r, m	(r) <- (m)	01000	m_R	m_C	r
	ST	m, r	(m) <- (r)	11000	m_R	m_C	r
	MOV	@r, m	if MPE = 1 : (MP, (r)) <- (m) if MPE = 0 : (BANK, m_R, (r)) <- (m)	01010	m_R	m_C	r
		m, @r	if MPE = 1 : (m) <- (MP, (r)) if MPE = 0 : (m) <- (BANK, m_R, (r))	11010	m_R	m_C	r
		m, #n4	(m) <- n4	11101	m_R	m_C	n4
	MOVT	DBF, @AR	SP <- SP – 1, ASR <- PC, PC <- AR, DBF <- (PC), PC <- ASR, SP <- SP + 1	00111	000	0001	0000
	PUSH	AR	SP <- SP – 1, ASR <- AR	00111	000	1101	0000
	POP	AR	AR <- ASR, SP <- SP + 1	00111	000	1100	0000
	PEEK	WR, rf	WR <- (rf)	00111	rf_R	0011	rf_C
	POKE	rf, WR	(rf) <- WR	00111	rf_R	0010	rf_C
	GET	DBF, p	DBF <- (p)	00111	p_H	1011	p_L
	PUT	p, DBF	(p) <- (DBF)	00111	p_H	1010	p_L

Instruction set	Mnemonic	Operand(s)	Operation	Machine code			
				Operation code	Operand(s)		
Branch	BR	addr	$PC_{10–0}$ <- addr, PAGE <- 0	01100	addr		
			$PC_{10–0}$ <- addr, PAGE <- 1	01101			
			$PC_{10–0}$ <- addr, PAGE <- 2	01110			
			$PC_{10–0}$ <- addr, PAGE <- 3	01111			
		@AR	PC <- AR	00111	000	0100	0000
Subroutines	CALL	addr	SP <- SP – 1, ASR <- PC, $PC_{12, 11}$ <- 0, $PC_{10–0}$ <- addr	11100	addr		
		@AR	SP <- SP – 1, ASR <- PC, PC <- AR	00111	000	0101	0000
	SYSCAL	entry	SP <- SP – 1, ASR <- PC, SGR <- 1, $PC_{12, 11}$ <- 0, $PC_{10–8}$ <- $entry_H$, $PC_{7–4}$ <- 0, $PC_{3–0}$ <- $entry_L$	00111	$entry_H$	0000	$entry_L$
	RET		PC <- ASR, SP <- SP + 1	00111	000	1110	0000
	RETSK		PC <- ASR, SP <- SP + 1 and skip	00111	001	1110	0000
	RETI		PC <- ASR, INTR <- INTSK, SP <- SP + 1	00111	100	1110	0000
Interrupt	EI		INTEF <- 1	00111	000	1111	0000
	DI		INTEF <- 0	00111	001	1111	0000
Others	STOP	s	STOP	00111	010	1111	s
	HALT	h	HALT	00111	011	1111	h
	NOP		No operation	00111	100	1111	0000

20.2 OPERAND CODING RULES

This section explains the rules governing the coding of operands.

The following rules are applied to an operand type check.

When multiple operands are specified, the types of all the specified operands are checked simultaneously. If an operand type error occurs, the error is classified into one of the following types. The error message output when an operand type is invalid thus depends on the error type.

- **When the types of all operands are invalid**
  The message "F011: Illegal first operand type" is output.

- **When two operands are specified, the first of which is invalid**
  The message "F011: Illegal first operand type" is output.

- **When two operands are specified, the second of which is invalid**
  The message "F012: Illegal second operand type" is output.

- **When all operand types are valid, but the operand pattern cannot be recognized**
  The message "F192: Illegal operand type" is output.

If an operand of an invalid type is specified, an error message is output according to the above rules.

20.2.1 Operand (r)

[Explanation]

(1) Specify an <expression (MEM type)>. If other than an MEM-type <expression> is specified, an error occurs.

(2) An <expression> of other than MEM type can be specified by using the type conversion function.

(3) An address cannot be directly specified with a numeric. If the user attempts to directly specify an address with a numeric, an error occurs. ".MD.11H" can be specified, however.

(4) If the specified address points to uninstalled memory, an error does not occur. If a data memory address above 7FH is specified, an error occurs. For the first operand, the message "F014: Illegal first operand value" is output. For the second operand, the message "F015: Illegal second operand value" is output.

(5) A symbol with the multi-nibble attribute can also be specified. When such a symbol is specified, any nibble information is ignored.

(6) The row address of the <expression>, specified for the first operand, and the value of the register pointer (RP) are not checked.

[Examples]

(1) Example of correct use

```
M011    MEM    0.11H
R00     MEM    0.00H
        ADD    R00,M011    An MEM-type symbol is specified for operand (r).
```

(2) When a numeric is specified directly

```
M011    MEM    0.11H
        ADD    00H,M011    An error (F011: Illegal first operand type) occurs because DAT-
                           type data is specified for operand (r) (the system determines that a
                           DAT-type numeric has been directly specified).
```

(3) When a numeric is specified directly

```
M011    MEM     0.11H
        ADD     .MD.00H,M011    An error does not occur because the type is converted.
```

(4) When the system cannot determine whether the specified data is a numeric

```
M011    MEM     0.11H
        ADD     0.00H,M011      An error (F037:  Syntax error) occurs because the system
                                cannot determine whether the data specified for operand (r) is
                                an <expression>.
```

20.2.2 Operand (m)

[Explanation]

(1) Specify an <expression (MEM type)>. If other than an MEM-type <expression> is specified, an error occurs.

(2) An <expression> of other than MEM type can also be specified using the type conversion function.

(3) An address cannot be directly specified with a numeric. If the user attempts to directly specify an address with a numeric, an error occurs. ".MD.11H" can be specified, however.

(4) If the specified address points to uninstalled memory, an error does not occur. If a data memory address above 7FH is specified, an error occurs. For the first operand, the message "F014: Illegal first operand value" is output. For the second operand, the message "F015: Illegal second operand value" is output.

(5) A symbol with the multi-nibble attribute can also be specified. When such a symbol is specified, any nibble information is ignored.

[Examples]

(1) Example of correct use

```
M011    MEM    0.11H
R00     MEM    0.00H
        ADD    R00,M011        An MEM-type symbol is specified for operand (m).
```

(2) When a numeric is specified directly

```
R00     MEM    0.00H
        ADD    R00,11H         An error (F012: Illegal second operand type) occurs because
                               DAT-type data is specified for operand (m) (the system
                               determines that a DAT-type numeric has been directly
                               specified).
```

(3) When a numeric is specified directly

```
R00     MEM    0.00H
        ADD    R00,.MD.11H     An error does not occur because the type is converted.
```

(4) When the system cannot determine whether the specified data is a numeric

```
R00     MEM     0.00H
        ADD     R00,0.00H
```

An error (F037: Syntax error) occurs because the system cannot determine whether the data specified for operand (m) is an <expression>.

20.2.3 Operand (#n4)

[Explanation]

(1) Specify an <expression (DAT type)>, preceded by "#." If other than a DAT-type <expression> is specified, an error occurs.

(2) If "#" is omitted, an error (F037: Syntax error) occurs.

(3) If only "#" is specified, without an <expression>, an error (F037: Syntax error) occurs.

(4) If the evaluation value is other than 0 to 15, an error (F015: Illegal second operand value) occurs.

(5) A numeric can also be specified directly.

20.2.4 Operand (AR)

[Explanation]

(1) An <expression> can be specified. Normally, specify reserved word symbol AR. A DAT-type <expression> for which the evaluation value is 40H can also be specified. This is because, for an operand, only the type and evaluation value are checked.

(2) If a non-resolved symbol is specified for the <expression>, an error (F058: Undefined symbol) occurs.

(3) If an external symbol is specified for the <expression>, an error (F150: Impossible to write the external symbol) occurs.

(4) If the type obtained when the <expression> is evaluated is other than DAT, an error occurs.

(5) If the evaluation value of the <expression> is other than 40H, an error (F014: Illegal first operand value) occurs.

20.2.5 Operand (IX)

[Explanation]

(1) An <expression> can be specified. Normally, specify reserved word symbol IX. A DAT-type <expression> having an evaluation value of 01H can also be specified. This is because, for an operand, only the type and evaluation value are checked.

(2) If a non-resolved symbol is specified for the <expression>, an error (F058: Undefined symbol) occurs.

(3) If an external symbol is specified for the <expression>, an error (F150: Impossible to write the external symbol) occurs.

(4) If the type obtained when the <expression> is evaluated is other than DAT, an error occurs.

(5) If the evaluation value of the <expression> is other than 01H, an error (F014: Illegal first operand value) occurs.

20.2.6 Operand (@r)

[Explanation]

(1) Specify an <expression (MEM type)>, preceded by "@." If other than an MEM-type <expression> is specified, an error occurs.

(2) If "@" is omitted, an error (F037: Syntax error) occurs.

(3) If only "@" is specified, without an <expression>, an error (F037: Syntax error) occurs.

(4) An <expression> of other than MEM type can also be specified using the type conversion function.

(5) Addresses cannot be directly specified with a numeric. If the user attempts to directly specify an address with a numeric, an error occurs. ".MD.11H" can be specified, however.

(6) If the specified address points to uninstalled memory, an error does not occur. If a data memory address above 7FH is specified, an error occurs. For the first operand, the message (F014: Illegal first operand value) is output. For the second operand, the message (F015: Illegal second operand value) is output.

(7) A symbol with the multi-nibble attribute can also be specified. When such a symbol is specified, any nibble information is ignored.

20.2.7 Operand (DBF)

[Explanation]

(1) An <expression> can be specified. Normally, specify reserved word symbol DBF. A DAT-type <expression> for which the evaluation value is 0FH can also be specified. This is because, for an operand, only the type and evaluation value are checked.

(2) If a non-resolved symbol is specified for the <expression>, an error (F058: Undefined symbol) occurs.

(3) If an external symbol is specified for the <expression>, an error (F150: Impossible to write the external symbol) occurs.

(4) If the type obtained when the <expression> is evaluated is other than DAT, an error occurs.

(5) If the evaluation value of the <expression> is other than 0FH, an error (F014: Illegal first operand value) occurs.

20.2.8 Operand (@AR)

[Explanation]

(1) The format is @<expression>.

(2) If a non-resolved symbol is specified for the <expression>, an error (F058: Undefined symbol) occurs.

(3) If an external symbol is specified for the <expression>, an error (F150: Impossible to write the external symbol) occurs.

(4) If the type obtained when the <expression> is evaluated is other than DAT, an error occurs.

(5) If the evaluation value of the <expression> is other than 40H, an error (F014: Illegal first operand value) occurs for the first operand. Or, for the second operand, (F015: Illegal second operand value) occurs.

20.2.9 Operand (WR)

[Explanation]

(1) An <expression> can be specified. Normally, specify reserved word symbol WR. An MEM-type <expression> for which the evaluation value is 0.78H can also be specified. This is because, for an operand, only the type and evaluation value are checked.

(2) If a non-resolved symbol is specified for the <expression>, an error (F058: Undefined symbol) occurs.

(3) If an external symbol is specified for the <expression>, an error (F150: Impossible to write the external symbol) occurs.

(4) If the type obtained when the <expression> is evaluated is other than MEM, an error occurs.

(5) If the evaluation value of the <expression> is other than 0.78H, an error (F014: Illegal first operand value) occurs for the first operand. Or, for the second operand, (F015: Illegal second operand value) occurs.

20.2.10 Operand (rf)

[Explanation]

(1) Specify an <expression (MEM type)>. If other than an MEM-type <expression> is specified, an error occurs.

(2) An <expression> of other than MEM type can also be specified by using the type conversion function.

(3) Addresses cannot be directly specified with a numeric. If the user attempts to directly specify an address with a numeric, an error occurs. ".MD.81H" can be specified, however.

(4) If a write-only register file is specified for the operand of the PEEK instruction, an error occurs.
If a read-only register file is specified for the operand of the POKE instruction, an error occurs.
If an unused area is specified with the PEEK or POKE instruction, an error occurs.

(5) A symbol with the multi-nibble attribute can also be specified. When such a symbol is specified, any nibble information is ignored.

(6) If the evaluation value of the <expression> is other than 40H to BFH, an error occurs.

20.2.11 Operand (p)

[Explanation]

(1) Specify an <expression (DAT type)>. If other than a DAT-type <expression> is specified, an error occurs.

(2) An <expression> of other than DAT type can also be specified using the type conversion function.

(3) A numeric can also be specified directly.

(4) If a write-only port is specified for the operand of the GET instruction, an error occurs.
If a read-only port is specified for the operand of the PUT instruction, an error occurs.
If an unused address is specified with the GET or PUT instruction, an error occurs.

(5) If the evaluation value of the <expression> is greater than 7FH, an error (F014: Illegal first operand value) occurs for the first operand. Or, for the second operand, (F015: Illegal second operand value) occurs.

(6) A symbol with the multi-nibble attribute can also be specified. When such a symbol is specified, any nibble information is ignored.

20.2.12 Operand (#n)

[Explanation]

(1) Specify an <expression (DAT type)> preceded by "#." If other than a DAT-type <expression> is specified, an error occurs.

(2) If "#" is omitted, an error (F037: Syntax error) occurs.

(3) If only "#" is specified, without an <expression>, an error (F037: Syntax error) occurs.

(4) If the evaluation value is other than 0 to 15, an error (F015: Illegal second operand value) occurs.

(5) A numeric can be directly specified for the <expression>.

20.2.13 Operand (addr)

[Explanation]

(1) Specify an <expression (LAB type)>. If other than an LAB-type <expression> is specified, an error occurs.

(2) An <expression> of other than LAB type can also be specified using the type conversion function.

(3) Addresses cannot be directly specified with a numeric. If the user attempts to directly specify an address with a numeric, an error occurs. ".LD.11H" can be specified, however.

(4) If the specified address points to uninstalled memory (neither ROM area nor EPA area), an error (F152: The address is out of ROM) occurs.

20.2.14 Operand (entry)

[Explanation]

(1) Specify an <expression (DAT type)>. If other than a DAT-type <expression> is specified, an error occurs.

(2) An <expression> of other than DAT type can also be specified using the type conversion function.

(3) A numeric can also be specified directly.

(4) If the evaluation value of the <expression> is greater than 7FH, an error (F014: Illegal first operand value) occurs.

20.2.15 Operand (s)

[Explanation]

(1) Specify an <expression (DAT type)>. If other than a DAT-type <expression> is specified, an error occurs.

(2) An <expression> of other than DAT type can also be specified using the type conversion function.

(3) A numeric can also be specified directly.

(4) If the evaluation value of the <expression> is greater than 0FH, an error (F014: Illegal first operand value) occurs.

20.2.16 Operand (h)

[Explanation]

(1) Specify an <expression (DAT type)>. If other than a DAT-type <expression> is specified, an error occurs.

(2) An <expression> of other than DAT type can also be specified using the type conversion function.

(3) A numeric can also be specified directly.

(4) If the evaluation value of the <expression> is greater than 0FH, an error (F014: Illegal first operand value) occurs.

CHAPTER 21 BUILT-IN MACRO INSTRUCTIONS

RA17K supports built-in macro instructions, thus freeing the user from the need to define such instructions before using them.

This chapter explains the following built-in instructions, as supported by RA17K:

- System register operation instructions
- Flag (bit) manipulation instructions
- Instructions extended from 17K series instructions
- Structured instructions for creating structured program descriptions.

The built-in macro instructions can be classified into four types.

* • **System register operation instructions**

BANKn	SETBANK	SETRP	SETMP	SETIX	SETAR

• **Flag operation instructions**

SETn	CLRn	NOTn	SKTn	SKFn	INITFLG

* • **Extended instructions**

SETX	CLRX	NOTX	SKTX	SKFX	INITFLGX
MOVX	MOVTX	ADDX	ADDCX	ADDSX	ADDCSX
SUBX	SUBCX	SUBSX	SUBCSX	SKEX	SKNEX
SKGEX	SKGTX	SKLEX	SKLTX	RORCX	ROLCX
SHRX	SHLX	ANDX	ORX	XORX	BRX
CALLX	SYSCALX				

* • **Structured instructions**

_IF..._ELSEIF..._ELSE..._ENDIF
_WHILE..._ENDW
_SWITCH..._CASE..._DEFAULT..._ENDS
_REPEAT..._UNTIL
_FOR..._NEXT
_BREAK
_CONTINUE
_GOTO

[Notes]

(1) A <label> can be specified for all built-in macros (excluding structured instructions). If an invalid <label> is specified, an error (see **Section 4.3**) occurs. In this case, the built-in macro is not expanded, an NOP instruction being generated instead.

(2) A symbol specified for an operand must be a backward reference symbol. Neither forward reference symbols nor external symbols can be specified. If a forward reference symbol is specified, an error (F058: Undefined symbol) occurs. If an external symbol is specified, an error (F150: Impossible to write the external symbol) occurs. In this case, the built-in macro is not expanded, an NOP instruction being generated instead.

[Label in a built-in macro]

Expanding a built-in macro may cause a branch instruction, such as a BR instruction, to be generated. At this time, a label is automatically created for the branch destination and a branch instruction is created for the label.

Label rule applied to built-in macro expansion

`?Lxxxx` xxxx : Decimal serial number (starting with 0)

[Example]

```
SKT3   FLG1 , FLG2 , FLG3
       ↓

       SKT      .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
       BR       ?L0
       SKF      .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
       SKT      .MF.FLG2 SHR 4 , #.DF.FLG2 AND 0FH
?L0
```

[Built-in macro expansion]

Built-in macro expansion varies depending on whether the previous instruction has a skip function.

(1) When the previous instruction has no skip function

-> The built-in macro is expanded to the instructions required to perform processing.

(2) When the previous instruction has a skip function

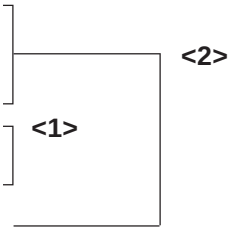
-> The built-in macro is expanded so that the entire macro can be skipped by the skip function. When the built-in macro instruction is expanded to multiple instructions, therefore, instructions which skip those instructions are also generated.

[Example 1]

```
M1      MEM     1.10H
M2      FLG     1.20H.1
M3      FLG     1.34H.2

        SKE     M1,#1
        SET2    M2,M3
        ↓

        SKE     M1,#1
        BR      ?L0
        BR      ?L1
?L0:
        OR      .MF.M2 SHR 4 , #.DF.M2 AND 0FH
        OR      .MF.M3 SHR 4 , #.DF.M3 AND 0FH
?L1:
```



<1> Expansion of SET2 M2,M3

<2> Instructions are generated because the SET2 instruction follows an instruction having a skip function

* 21.1 SYSTEM REGISTER OPERATION INSTRUCTIONS

The system register operation instructions are built-in macro instructions used to set values in the system registers allocated in data memory.

21.1.1 BANKn

[Format]

```
[<label>:][Δ]BANKn[Δ][;<comment>]
```

n: 0 ≤ n ≤ 15, n is a decimal integer.

[Function]

BANKn sets the value indicated by n in the bank register (address 79H in data memory).
n indicates a bank number.

[Explanation]

(1) The value set by BANKn can be checked using ZZZBANK.

* (2) There is no functional difference between BANKn and SETBANK.

[Notes]

(1) The BANKn instruction cannot be applied to a device having no bank register. Only the BANK0 instruction can be used for a device for which the bank register is fixed to 0.

(2) The value indicated by n is the number of a bank installed in the target device. If a number corresponding to an uninstalled bank is specified, an error (F046: Invalid BANK No.) occurs and an NOP instruction is generated. If n indicates a value of more than 15, an error (F037: Syntax error) occurs.

(3) If an operand is specified with the BANK instruction, an error (F037: Syntax error) occurs. That line is invalidated.

[Sample expansion]

BANKn is expanded to the following instruction, according to the value specified with n:

```
BANK3
  ↓
  MOV   BANK , #03
```

21.1.2 SETBANK

[Format]

```
[<label>:][Δ]SETBANKΔ<expression (MEM type)>[Δ][;<comment>]
```

[Function]

SETBANK sets the bank information for <expression (MEM type)> specified as the operand in the bank register.

[Explanation]

(1) The value set by SETBANK can be checked using ZZZBANK.

(2) There is no functional difference between BANKn and SETBANK.

[Notes]

(1) Only one <expression> can be specified for the operand. If less than or more than one <expression> is specified, an error (F037: Syntax error) occurs, and an NOP instruction is generated as an object.

(2) If the type obtained when the <expression> is evaluated is other than MEM, an error (F045: Invalid type) occurs, and an NOP instruction is generated as an object.

(3) If the bank number obtained when the <expression> is evaluated is not used with the product, an error (F046: Invalid BANK No.) occurs, and an NOP instruction is generated.

[Sample expansion]

An MEM-type symbol is specified as the operand.

```
MEM1    MEM 1.23H     ;Symbol definition (Bank =1, Row = 2, Column = 3)
        SETBANK       MEM1
        ↓
        MOV           BANK,#.DM. (MEM1)SHR 8 AND 0FH  ;BANK1
```

21.1.3 SETRP

[Format]

```
[<label>:][Δ]SETRPΔ<expression (MEM type)>[Δ][;<comment>]
```

[Function]

SETRP sets the bank and row information of <expression (MEM type)> specified as the operand in RPH and RPL.

[Notes]

(1) Only one <expression> can be specified as the operand. If less than or more than one <expression> is specified, an error (F037: Syntax error) occurs, and an NOP instruction is generated as an object.

(2) If the type obtained when the <expression> is evaluated is other than MEM, an error (F045: Invalid type) occurs, and an NOP instruction is generated as an object.

(3) If the bank number obtained when the <expression> is evaluated is not used with the product, an error (F046: Invalid BANK No.) occurs, and an NOP instruction is generated.

[Sample expansion]

Only a symbol is specified as the operand.

```
MEM1   MEM    1.23H ;Symbol definition (Bank =1, Row = 2, Column = 3)

       SETRP  MEM1
       ↓
       MOV    RPH,#.DM.(MEM1)SHR 8 AND 0FH      ;RPH=1 (bank)
       AND    RPL,#01H
       OR     RPL,#.DM.(MEM1)SHR 3 AND 0EH      ;RPH=2 (row)
```

21.1.4 SETMP

[Format]

```
[<label>:][Δ]SETMPΔ<expression (MEM type)>[Δ][;<comment>]
```

[Function]

SETMP sets the bank and row information of <expression (MEM type)> specified as the operand in MPH and MPL.

[Notes]

(1) Only one <expression> can be specified as the operand. If less than or more than one <expression> is specified, an error (F037: Syntax error) occurs, and an NOP instruction is generated as an object.

(2) If the type obtained when the <expression> is evaluated is other than MEM, an error (F045: Invalid type) occurs, and an NOP instruction is generated as an object.

[Sample expansion]

```
MEM1   MEM 1.23H   ;Symbol definition (Bank =1, Row = 2, Column = 3)

       SETMP  MEM1
       ↓
       AND    MPH,#08H
       OR     MPH,#.DM.(MEM1)SHR 9 AND 07H
       MOV    MPL,#(.DM.(MEM1)SHR 4 AND 0FH)OR(.DM.(MEM1)SHR 5 AND 08H)
```

21.1.5 SETIX

[Format]

```
[<label>:][Δ]SETIXΔ<expression (MEM type)>[Δ][;<comment>]
```

[Function]

SETIX sets the bank, row, and column information of <expression (MEM type)> specified as the operand in IXH, IXM, and IXL.

[Notes]

(1) Only one <expression> can be specified as the operand. If less than or more than one <expression> is specified, an error (F037: Syntax error) occurs, and an NOP instruction is generated as an object.

(2) If the type obtained when the <expression> is evaluated is other than MEM, an error (F045: Invalid type) occurs, and an NOP instruction is generated as an object.

[Sample expansion]

```
MEM1   MEM    1.23H ;Symbol definition (Bank =1, Row = 2, Column = 3)

       SETIX  MEM1
       ↓
       AND    IXH,#08H
       OR     IXH,#.DM.(MEM1)SHR 9 AND 07H
       MOV    IXM,#(.DM.(MEM1)SHR 4 AND 07H)OR(.DM.(MEM1)SHR 5 AND 08H)
       MOV    IXL,#.DM.(MEM1)AND 0FH
```

21.1.6 SETAR

[Format]

```
[<label>:][Δ]SETARΔ<expression (LAB type)>[Δ][;<comment>]
```

[Function]

SETAR sets the evaluation value of <expression (LAB type)> specified as the operand in AR.

[Notes]

(1) Only one <expression> can be specified as the operand. If less than or more than one <expression> is specified, an error (F037: Syntax error) occurs, and an NOP instruction is generated as an object.

(2) If the type obtained when the <expression> is evaluated is other than LAB, an error (F045: Invalid type) occurs, and an NOP instruction is generated as an object.

[Sample expansion]

```
MEM1   LAB    1.23H ;Symbol definition (Bank =1, Row = 2, Column = 3)

       SETAR LAB1
       ↓
       MOV    AR3,#.DL.(LAB1)SHR 12 AND 0FH
       MOV    AR2,#.DL.(LAB1)SHR 8 AND 0FH
       MOV    AR1,#.DL.(LAB1)SHR 4 AND 0FH
       MOV    AR0,#.DL.(LAB1)AND 0FH
```

21.2 FLAG OPERATION INSTRUCTIONS

The 17K series does not support any instructions that enable the manipulation of data bit-by-bit. RA17K implements simulated flag (bit) operation using flag-type symbols and built-in macro instructions for flag manipulation.

When manipulating multiple flags (bits) located at the same address, the processing can be achieved using fewer instructions than is required when manipulating flags located at different addresses. The number of steps required for manipulating multiple flags varies greatly depending on the relationships between the addresses at which the flags are located.

To set four flags having symbolic names A, B, C, and D, when all four are located at the same data memory address, the processing can be achieved only one OR instruction.

[Example 1] When A, B, C, and D are assigned to bits 3, 2, 1, and 0 at the same address (0.43H)

```
(MEM043    MEM    0.43H)
           OR     MEM043 , #1111B
```

If these four flags were to be located at different addresses, four OR instructions would be required.

[Example 2] When A, B, C, and D are defined as the following symbols:

```
A    FLG    0.10H.0
B    FLG    0.11H.1
C    FLG    0.12H.2
D    FLG    0.13H.3
```

To set each bit, the following four instructions are required:

```
OR     .MF.A SHR 4 , #0001B
OR     .MF.B SHR 4 , #0010B
OR     .MF.C SHR 4 , #0100B
OR     .MF.D SHR 4 , #1000B
```

When the user codes a source program for manipulating multiple flags, the user may have to consider the bit locations of the flags defined as symbols and determine whether one instruction or multiple instructions are required. Coding a program in this way not only reduces coding efficiency, but may also lead to the introduction of a bug. When there is a source-level library whose operation is guaranteed, library compatibility cannot be maintained for any devices having internal flags located at different addresses.

The use of flag operation instructions can help to eliminate such inconveniences when coding a source program.

The following built-in macro instructions are provided for flag manipulation:

SETn : Sets flags.
CLRn : Resets flags.
NOTn : Reverses the contents of flags.
SKTn : Tests the contents of flags: When 1 is returned, skips the next instruction.
SKFn : Tests the contents of flags: When 0 is returned, skips the next instruction.
INITFLG : Initializes flags.

n indicates the number of flags specified in the operand field. For example, to set four flags, A, B, C, and D, specify the following macro:

* Extended instructions SETX to SKFX determine the number of symbols that are automatically specified as the operand for replacement with SETn to SKFn. So, the functions are the same as SETn to SKFn.

[Example 3]
```
SET4    A , B , C , D
or
SETX    A , B , C , D
```

For the above macro, RA17K obtains the addresses and bit positions, using the evaluation values of flags A, B, C, and D, then automatically generates the optimum object. (The optimum object is the shortest object.) When flags A, B, C, and D are all located at the same address, RA17K expands the macro to the object shown in [Example 1]. When the flags are located at different addresses, RA17K expands the macro to the object shown in [Example 2]. Because RA17K expands the macro, automatically determining whether the flags specified in the operand field are data memory flags or register file flags, the user need not consider the address space in which the flags to be manipulated are mapped.

Because symbolic names are specified in the operand field of each built-in macro, when a flag definition location (address or bit location) is changed during program coding, the user simply has to change the symbol definition. There is no need to modify the program itself. By defining the same symbolic name for each flag, the user does not have to consider the flag location when reusing another program routine. The use of this method enables the creation of a program library.

[INITFLG]

The following explains the processing performed when flags A, B, C, and D are assigned to the same address, with A and B initialized to 1, and C and D initialized to 0.

When flags A, B, C, and D are assigned to bits 3, 2, 1, and 0 at the same address (1.53H), initialization of the four flags with the 17K series requires only the following instruction:

[Example 4]

```
(MEM153     MEM     1.53H)
          MOV     MEM153 , #1100B
```

To initialize the flags using built-in macro instructions such as SETn and CLRn, the user would have to specify the following instructions:

[Example 5]

```
SET2    A,B
CLR2    C,D
   ↓
OR      MEM153 , #1100H
AND     MEM153 , #1100H
```

The above example illustrates that a redundant object may be generated when built-in macro instructions merely have functions for setting and resetting flags. The user may also specify an MOV instruction to reduce the number of steps, ignoring program compatibility or the possibility of flag relocation. In this case, of course, relocation of the flags initialized using the MOV instruction cannot be guaranteed. This is because RA17K is incapable of recognizing that the MOV instruction is used to initialize flags after a flag location has been changed after program coding.
To overcome this problem, RA17K provides the INITFLG instruction.
The following INITFLG instruction has the same effect as the above example:

[Example 6]

```
INITFLG       A ,B ,NOT C,NOT D
```

When all the flags are located at the same address, this macro is expanded to one instruction as shown in [Example 4]. When at least one bit is located at a different address, however, this macro is expanded to the optimum instructions.

[Example 7] When D is located at an address (1.54H) other than that at which the other flags (A, B, and C) are located

```
INITFLG        A , B , NOT C , NOT D
   ↓
CLR1    C
        AND    MEM153 , #1101B
SET2    A , B
        OR     MEM153 , #1100B
CLR1    D
        AND    MEM154 , #1110B
```

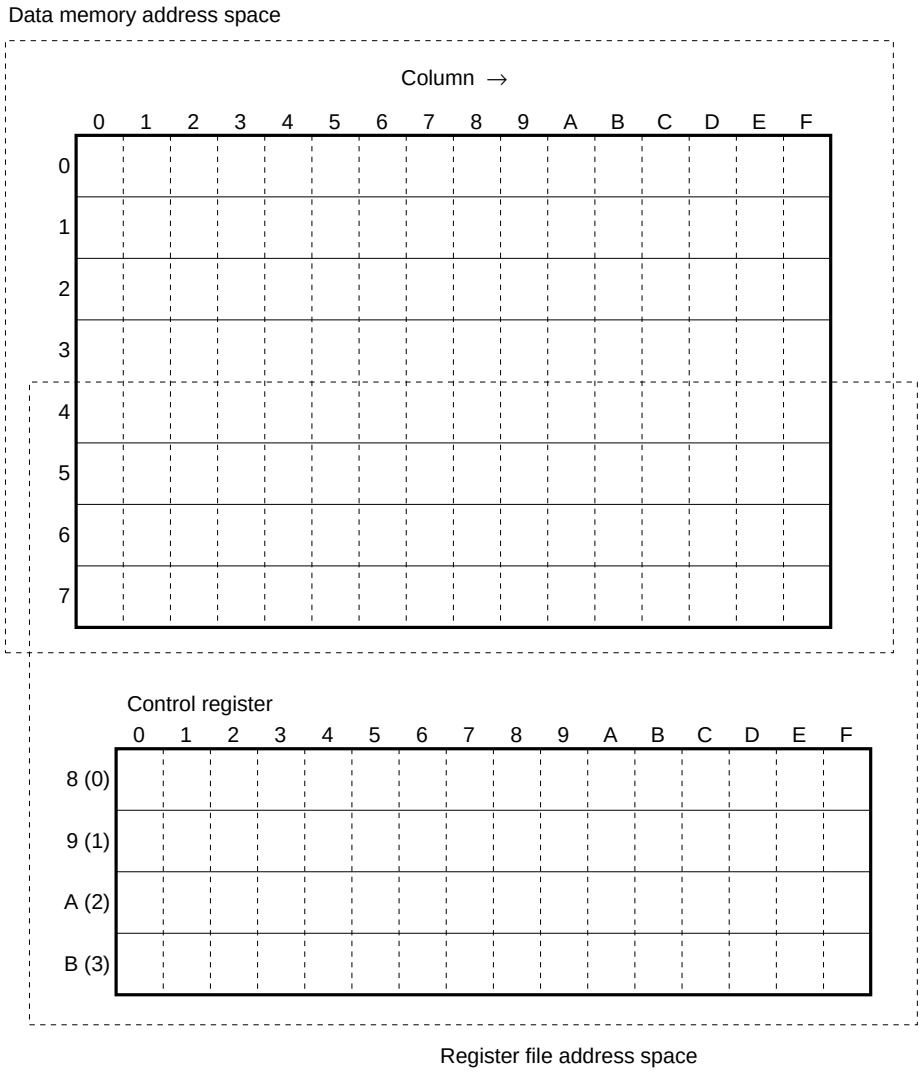
As shown above, the macro is expanded to three instructions.

In the above explanation, data memory flags are used as examples. For register file flags, the format is exactly the same. For an explanation of the register file, see the following item.

[Concept of register file]

The register file (RF) consists of registers that are mainly used for controlling hardware components peripheral to the CPU. It has a capacity of 4 bits x 128 nibbles. In the register file space, peripheral hardware control registers (control registers) are assigned to the 64 high-order nibbles (00H to 3FH). Addresses 40H to 7FH of the currently specified bank in data memory are assigned to the 64 low-order nibbles (40H to 7FH). Addresses 40H to 7FH in each bank of data memory are located in both the data memory address space and register file space.

The register file address space is shown in the following memory map image.



Control registers in the register file space can be accessed from any bank. The evaluation value for the bank is set to 0, however. Addresses 00H to 3FH are physically assigned to the control registers. However, the evaluation value for the address of each control register indicates the value obtained by adding 1 to the most significant bit (80H to 0BFH), thus distinguishing the control register space from the data memory space.

[Example 8] Stack pointer (SP) -> 0.81H

Each register in the register file space is accessed via the window register (WR [address 0.78H]). The following two instructions are used to access the register file space:

```
PEEK   WR, rf  (Transfers the contents of the register specified by rf to the WR.)
POKE   rf, WR  (Transfers the contents of the WR to the register specified by rf.)
```

Note that when a control register in the register file space is specified for an operand of the direct transfer instruction (MOV) or arithmetic/logical instruction (such as ADD or SUB), the object which manipulates the data memory area, at an address for which the most significant bit (bit 7) is masked, is generated.

[Flag operation for register file space]

RA17K enables the use of flag operation instructions by defining a control register in the register file space in the same way as for data memory. The user can manipulate a control register flag simply by specifying the control flag in the register file space in the operand field of a built-in macro. When specifying the flag, the user does not have to consider whether the specified flag is located in the data memory space or register file space. The assembler automatically generates the optimum object.

Assume that a flag with symbolic name DMF is defined for bit 1 at address 32H in data memory, and that a flag with symbolic name RFF is defined for bit 2 at address 25H (= 0.0A5H) in the register file. The built-in macro for setting the contents of each flag is expanded as follows:

[Example 9]

```
<1>  SET1        DMF
          ↓
         OR      MEM032,#0010B

<2>  SET1        RFF
          ↓
         PEEK    WR , MEM0A5
         OR      WR , #0100B
         POKE    MEM0A5 , WR
```

When this operation is specified using the SET2 instruction, the instruction is expanded to two SET1 instructions. These two SET1 instructions are expanded to the instructions shown in [Example 9].

[Example 10]

```
SET2        DMF , RFF
  ↓
  SET1   DMF
         OR       MEM032 ,#0010B
  SET1   RFF
         PEEK     WR ,MEM0A5
         OR       WR ,#0100B
         POKE     MEM0A5 , WR
```

[Optimization of flag operation for the register file space]

RA17K automatically determines whether each flag specified in the operand field of a flag operation instruction is located in the data memory space or register file space, then generates an object. It checks the most significant bit (bit 7) of the address using the evaluation value of the flag. When the bit is 0, RA17K recognizes the flag as being located in data memory. When the bit is 1, it recognizes the flag as being in the register file. For a flag in the register file space, the PEEK and POKE instructions are added as shown in the above item.

Control registers in the register file space are used for controlling peripheral hardware. The control flag names (words reserved for the target device) and addresses are defined in the device file. If an address is not defined in the device file, RA17K outputs an error message (F014: Illegal first operand value) for the PEEK and POKE instructions at expansion.

For example, the instruction for setting control flag RF330 (bit 0 at address 33H) in the register file space is expanded as follows:

[Example 11]

```
SET1        RF330
      ↓
      PEEK    WR , .MF.RF330
      OR      WR , #0001B
      POKE    .MF.RF330 , WR
```

When only flag RF330 is assigned to address 33H in the register file, the size of the object code can be reduced by one step for [Example 11] by coding the following instructions:

[Example 12]

```
MOV     WR,#0001B
POKE    .MF.RF330 ,WR
```

When all the flags assigned to an address are specified in the operand field of a flag operation instruction, the PEEK instruction is not required (when the flags assigned to a 1-nibble area constitute less than four bits, the instruction also is not required).
To enable this, the device file also contains flag map information in addition to the symbolic names and address information for the control flags. Flag map information indicates the number of flags and flag locations assigned to each 1-nibble area. By referencing this flag map information, RA17K can automatically expand the following built-in macro instructions without generating the PEEK instruction, as shown in [Example 12].

- SETn
- CLRn
- INITFLG

Using this function, the SET1 instruction shown in [Example 11] is automatically expanded as shown in [Example 12].
When the flags to be initialized using the INITFLG instruction are located at the same address, and no other flag is assigned to the 1-nibble area, the flags are processed by the MOV instruction as shown in [Example 13], below.

[Example 13] When three flags F0, F1, and F2 (bits 0, 1, and 2) in the register file space are to be initialized to 1, 1, and 0, respectively

- **When another flag is assigned to bit 3**

```
INITFLG       F0 , F1 , NOT F2
        ↓
   SET2       F0 , F1
        PEEK  WR , .MF.F0 SHR 4
        AND   WR , #1011B
        OR    WR , #1011B
        POKE  .MF.F0 SHR 4 , WR
```

- **When no other flag is assigned to bit 3**

```
INITFLG       F0 , F1 , NOT F2
        ↓
    MOV       WR , #0011B
    POKE      .MF.F0 SHR 4 , WR
```

* 21.2.1 SETn

[Format]

```
[<label>:][Δ]SETnΔ<FLG-type-expression-list>[Δ][;comment]
```

[Function]

SETn sets the flags specified in the operand field.
n indicates the number of flags in the operand field ($1 \leq n \leq 4$).

[Notes]

(1) If the value specified for n in the instruction differs from the number of flags specified in the operand field, an error (F037: Syntax error) occurs and the built-in macro is not expanded. In this case, an NOP instruction is generated as the object.

(2) If the type obtained when the <expression> is evaluated is of other than flag type, an error (F045: Invalid type) occurs and the built-in macro is not expanded. In this case, an NOP instruction is generated as the object.

(3) If multiple flags on different banks are specified in the operand field, an error (F063: Bank unmatch) occurs and the built-in macro is not expanded. In this case, an NOP instruction is generated as the object.
When both the control flags in the register file space and the flags in the data memory space are specified simultaneously, however, no error occurs provided the flags in the data memory space are in other than bank 0.

(4) When at least one control flag in the register file space is specified in the operand field, the contents of the window register (WR) are rewritten by the built-in macro instruction.
The user may include a built-in macro, for manipulating a control flag in the register file space, in an interrupt handling routine. In this case, save the contents of the WR in the data memory space at the beginning of the interrupt handling routine.

(5) If a value of 0 or 5 or greater is specified for n, an error (F037: Syntax error) occurs and an NOP instruction is generated as the object.

(6) If the system determines that a bank number which is not installed in the target device has been specified, upon evaluating the <expression>, an error (F046: Invalid BANK No.) occurs and an NOP instruction is generated as the object.

(7) If the system determines that a data memory address which does not exist in the target device has been specified upon evaluating the <expression>, an error (F067: Address error) occurs and an NOP instruction is generated as the object.

(8) If the system determines that the value of a bit location is other than 1, 2, 4, or 8, during the evaluation of the <expression>, an error (F044: Invalid value) occurs and an NOP instruction is generated as the object.

[Sample expansion]

SETn generates the shortest object for setting flags, using the evaluation values of the addresses of the flags specified in the operation field. An example of manipulating three bits in data memory is shown below:

[Example 1]

```
FLG1    FLG    3.3FH.1
FLG2    FLG    3.3FH.2
FLG3    FLG    3.30H.3

        SET1   FLG1                  ··<1>
        SET2   FLG1 , FLG2           ··<2>
        SET3   FLG1 , FLG2 , FLG3    ··<3>
```

For multiple flags located at the same address, SETn is expanded to one instruction.
<1>, **<2>**, and **<3>** are expanded as follows:

```
<1>     SET1   FLG1
               ↓
               OR     .MF.(FLG1) SHR 4, #.DF.(FLG1) AND 0FH

<2>     SET2   FLG1 , FLG2
               ↓
               OR     .MF.(FLG1) SHR 4, #.DF.((FLG1) OR (FLG2)) AND 0FH

<3>     SET3   FLG1 , FLG2 , FLG3
               ↓
               OR     .MF.(FLG1) SHR 4, #.DF.((FLG1) OR (FLG2)) AND 0FH
               OR     .MF.(FLG3) SHR 4, #.DF.(FLG1) AND 0FH
```

21.2.2 CLRn

[Format]

```
[<label>:][Δ]CLRnΔ<FLG-type-expression-list>[Δ][;<comment>]
```

[Function]

CLRn resets the flags specified in the operand field.
n indicates the number of flags in the operand field ($1 \leq n \leq 4$).

[Notes]

CLRn differs from SETn only in the object it generates.
The notes on specifying CLRn are the same as for SETn. See **[Notes]** in **Section 21.2.1**.

[Sample expansion]

CLRn generates the shortest object for resetting flags using the evaluation values of the addresses of the flags specified in the operation field. An example of manipulating three bits in data memory is shown below:

[Example 1]

```
FLG1    FLG     3.3FH.1
FLG2    FLG     3.3FH.2
FLG3    FLG     3.30H.3

        CLR1    FLG1                    ··<1>
        CLR2    FLG1 , FLG2             ··<2>
        CLR3    FLG1 , FLG2 , FLG3      ··<3>
```

For multiple flags located at the same address, CLRn is expanded to one instruction.
<1>, **<2>**, and **<3>** are expanded as follows:

```
<1>     CLR1    FLG1
                ↓
                AND     .MF.FLG1 SHR 4, #.DF.(NOT FLG1 AND 0FH)

<2>     CLR2    FLG1 , FLG2
                ↓
                AND     .MF.FLG1 SHR 4, #.DF.(NOT(FLG1 OR FLG2) AND 0FH)

<3>     CLR3    FLG1 , FLG2 , FLG3
                ↓
                AND     .MF.FLG1 SHR 4, #.DF.(NOT(FLG1 OR FLG2) AND 0FH)
                AND     .MF.FLG3 SHR 4, #.DF.(NOT FLG3 AND 0FH)
```

21.2.3 NOTn

[Format]

```
[<label>:][Δ]NOTnΔ<FLG-type-expression-list>[Δ][;<comment>]
```

[Function]

NOTn reverses the flags specified in the operand field (obtains their ones complement).
n indicates the number of flags in the operand fields ($1 \leq n \leq 4$).

[Notes]

(1) When all the control flags assigned to an address in the register file space are specified in the NOTn operand field, the PEEK instruction is not omitted, unlike in the case of SETn and CLRn. This is because the NOTn instruction reads the contents of flags, inverts their values, then transfers the inverted values to the flags.

(2) The notes on specifying NOTn are the same as for SETn. See **[Notes]** in **Section 21.2.1**.

[Sample expansion]

NOTn generates the shortest object for resetting flags, using the evaluation values of the addresses of the flags specified in the operation field. An example of manipulating three bits in data memory is shown below:

[Example 1]

```
FLG1    FLG    3.3FH.1
FLG2    FLG    3.3FH.2
FLG3    FLG    3.30H.3

        NOT1   FLG1                  ··<1>
        NOT2   FLG1 , FLG2           ··<2>
        NOT3   FLG1 , FLG2 , FLG3    ··<3>
```

For multiple flags located at the same address, NOTn is expanded to one instruction.
<1>, **<2>**, and **<3>** are expanded as follows:

```
<1>    NOT1   FLG1
              ↓
              XOR     .MF.FLG1 SHR 4, #.DF.FLG1 AND 0FH

<2>    NOT2   FLG1 , FLG2
              ↓
              XOR     .MF.FLG1 SHR 4, #.DF.(FLG1 OR FLG2) AND 0FH

<3>    NOT3   FLG1 , FLG2 , FLG3
              ↓
              XOR     .MF.FLG1 SHR 4, #.DF.(FLG1 OR FLG2) AND 0FH
              XOR     .MF.FLG3 SHR 4, #.DF.FLG1 AND 0FH
```

21.2.4 SKTn

[Format]

```
[<label>:][Δ]SKTnΔ<MEM-type-expression-list>[Δ][;<comment>]
```

[Function]

SKTn tests all the flags specified in the operand field. When all flags are set, SKTn skips the instruction following SKTn.

n indicates the number of flags in the operand field ($1 \leq n \leq 4$).

[Notes]

(1) When all the control flags assigned to an address in the register file space are specified in the SKTn operand field, the PEEK instruction is not omitted, unlike in the case of SETn and CLRn. This is because the SKTn instruction reads the contents of flags, then determines whether to skip the next instruction.

(2) The notes on specifying SKTn are the same as for SETn. See **[Notes]** in **Section 21.2.1**.

[Sample expansion]

SKTn generates the shortest object for checking flags, using the evaluation values of the addresses of the flags specified in the operation field.

The generated instruction string depends on the number of flags specified in the operand field, as well as their address locations, as follows:

[Example 1]

```
SKT1   FLG1                ··<1>
SKT2   FLG1 , FLG2         ··<2>
SKT3   FLG1 , FLG2 , FLG3  ··<3>
```

Assemble expansion (assuming that FLG1, FLG2, and FLG3 are all located in the data memory space)

<1> 1 flag

```
SKT1   FLG1
       ↓
       SKT    .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
```

<2> 2 flags

- **When FLG1 and FLG2 are located at the same address**

```
SKT2   FLG1 , FLG2
       ↓
       SKT      .MF.FLG1 SHR 4 , #.DF.(FLG1 OR FLG2) AND 0FH
```

- **When FLG1 and FLG2 are located at different addresses**

```
SKT2   FLG1 , FLG2
       ↓
       SKF      .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
       SKT      .MF.FLG2 SHR 4 , #.DF.FLG2 AND 0FH
```

<3> 3 flags

- **When FLG1, FLG2, and FLG3 are all located at the same address**

```
SKT3   FLG1 , FLG2 , FLG3
       ↓
       SKT      .MF.FLG1 SHR 4 , #.DF.((FLG1 OR FLG2) OR FLG3) AND 0FH
```

- **When FLG2 and FLG3 are located at the same address but FLG1 is located at a different address**

```
SKT3   FLG1 , FLG2 , FLG3
       ↓
       SKF      .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
       SKT      .MF.FLG1 SHR 4 , #.DF.(FLG2 OR FLG3) AND 0FH
```

- **When FLG1, FLG2, and FLG3 are all located at different addresses**

```
SKT3   FLG1 , FLG2 , FLG3
       ↓
       SKT      .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
       BR       ?L0
       SKF      .MF.FLG1 SHR 4 , #.DF.FLG1 AND 0FH
       SKT      .MF.FLG2 SHR 4 , #.DF.FLG2 AND 0FH
?L0:
```

21.2.5 SKFn

[Format]

```
[<label>:][Δ]SKFnΔ<FLG-type-expression-list>[Δ][;<comment>]
```

[Function]

SKFn tests all the flags specified in the operand field. When all the flags are reset, SKFn skips the instruction following SKFn.

n indicates the number of flags in the operand field ($1 \leq n \leq 4$).

[Notes]

(1) When all the control flags assigned to an address in the register file space are specified in the SKFn operand field, the PEEK instruction is not omitted, unlike in the case of SETn and CLRn. This is because the SKFn instruction reads the contents of flags, then determines whether to skip the next instruction.

(2) The notes on specifying SKFn are the same as for SETn. See **[Notes]** in **Section 21.2.1**.

[Sample expansion]

SKFn generates the shortest object for checking flags, using the evaluation values of the addresses of the flags specified in the operation field.

An instruction string is generated according to the number of flags specified in the operand field, as well as their address locations, in the same way as for the SKTn instruction. See **[Sample expansion]** in **Section 21.2.4**.

21.2.6 INITFLG

[Format]

```
[<label>:][Δ]INITFLGΔ<[NOT|INV]FLG-type-expression-list>[Δ][;<comment>]
```

[Function]

INITFLG generates the shortest object for initializing flags, using the evaluation values of the addresses of the flags specified in the operation field.

INITFLG initializes the flags, specified in the operand field and preceded by NOT or INV, to 0. Those preceded by nothing are initialized to 1. Up to four flags can be specified in the operand field.

As explained in **[Example 12]** in **Section 21.2**, when all the flags assigned to an address in the register file space are specified in the INITFLG operand field, the PEEK instruction is not generated.

[Notes]

(1) When control flags assigned to the same address in the register file space are to be initialized and all the flags assigned to that address are specified, the system automatically determines that the PEEK instruction need not be generated if these flags consist of fewer than four bits.

[Example]

Control flags TMCK0, TMCK1, and TMCK2 in the register file space are to be initialized. These flags are assigned to bits 0, 1, and 2 at the same address (no flag is assigned to bit 3).

To initialize these bits to 0, 1, and 1, respectively, the following INITFLG instruction is written:

```
INITFLG          NOT TMCK0 , TMCK1 , TMCK2
                ↓
         MOV     WR , #0110B
         POKE    .MF.TMCK0 , WR
```

(2) For those flags defined in the data memory space, when there are fewer than four bits to be initialized, INITFLG is not expanded to the MOV instruction.

[Example]

Flags in the data memory space are to be initialized.

```
FLG090    FLG      0.09H.0
FLG091    FLG      0.09H.1
FLG092    FLG      0.09H.2

INITFLG            FLG090 , NOT FLG091 , FLG092
          ↓
          CLR1     FLG091
                   AND      .MF.FLG091 SHR 4 , #.DF.(NOT FLG091 AND 0FH)
          SET2     FLG090 , FLG092
                   OR       .MF.FLG090 SHR 4 , #.DF.(FLG090 OR FLG092) AND 0FH
```

Note that INITFLG is not expanded as shown below:

```
INITFLG            FLG090 , NOT FLG091 , FLG092
          ↓
          MOV      .MF.FLG090 SHR 4 , #0101B
```

(3) The notes on specifying INITFLG are the same as for SETn. See **[Notes]** in **Section 21.2.1**.

* 21.3 EXTENDED INSTRUCTIONS

Extended instructions extend the number of nibbles for the symbols and numerics specified in the operand field of a 17K series instruction, and can represent multiple-nibble operations on one line. They improve coding efficiency and program readability. Users are encouraged to use extended instructions to maximize programming efficiency.

The following lists the built-in macros to which 17K series instructions are extended, as supported by version 2.xx of RA17K.

Table 21-1. Extended Instructions

Instruction		Reference page	Instruction		Reference page
Extended flag operation	SETX	p.330	Extended comparison	SKEX	p.365
	CLRX	p.333		SKNEX	p.367
	NOTX	p.335		SKGEX	p.369
	SKTX	p.337		SKGTX	p.371
	SKFX	p.339		SKLEX	p.373
	INITFLGX	p.341		SKLTX	p.375
Extended transfer	MOVX	p.343	Extended rotation	RORCX	p.377
	MOVTX	p.348		ROLCX	p.378
Extended addition	ADDX	p.349	Extended shift	SHRX	p.379
	ADDCX	p.351		SHLX	p.380
	ADDSX	p.353	Extended logic operation	ANDX	p.381
	ADDCSX	p.355		ORX	p.383
Extended subtraction	SUBX	p.357		XORX	p.385
	SUBCX	p.359	Extended branch	BRX	p.387
	SUBSX	p.361	Extended subroutine call	CALLX	p.388
	SUBCSX	p.363		SYSCALX	p.389

As explained above, the above instructions can be used to represent a function performed by combining multiple 17K series instructions with one built-in macro instruction. This improves program readability and maximizes programming efficiency because object optimization is performed.

[Memory-to-memory operations enabled]

Within an extended instruction, register pointers and bank numbers are managed to enable memory-to-memory operation.

If the MEM-type symbol information specified in the first operand does not match the current register pointer and bank number when an instruction is expanded, the SETBANK and SETRP built-in macros are executed using the MEM-type symbol to match the symbol specified in the first operand.

In addition, an operation involving a mixture of horizontal and vertical types is allowed. When a vertical-type symbol is specified in the first operand, operation is performed by updating the register pointer.

[Notes]

(1) When the number of nibbles in the first operand differs from the number of nibbles in the second operand, the following processing is performed.

[Number of nibbles in first operand = Number of nibbles in second operand]

Normal processing

[Number of nibbles in first operand > Number of nibbles in second operand]

Zeros are added to the value of the second operand to match the number of nibbles in the first operand.

[Number of nibbles in first operand < Number of nibbles in second operand]

A warning (W186: Nibble number unmatch. Unified the number to left) is output, and the number of nibbles in the second operand is reduced to match the number of nibbles in the first operand.

(2) For a memory-to-memory operation, a mismatch is allowed between the bank number defined by the MEM-type symbol specified in the first operand and the value specified in SETBANK or BANKn, and between a row address value and the value of the register pointer set in SETRP. If a mismatch is found, the assembler automatically generates an instruction (SETBANK or SETRP) for setting a bank number and register pointer.

The condition for generating each instruction is described below.

- Condition for generating the SETBANK instruction
  - The bank number defined by the symbol specified in the first operand differs from the current bank number set in SETBANK or BANKn.
  - When this instruction is expanded, the currently set bank number is unknown.

- Condition for generating the SETRP instruction
  - The row address defined by the symbol specified in the first operand differs from the current register pointer value set in SETRP.
  - When this instruction is expanded, the currently set register pointer value is unknown.

(3) When data consisting of multiple nibbles is processed, big endian mode is used.

21.3.1 SETX

[Format]

```
[<label>:][Δ]SETXΔ<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SETXΔ<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SETXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SETXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
```

[Function]

The flag specified in each operand field is set to 1.
In the operand fields, up to four FLG-type symbols can be specified.

[Notes]

(1) If the type obtained when an <expression> is evaluated is other than FLG, an error (F045: Invalid type) occurs. In this case, the line is invalidated, and an NOP instruction is generated.

(2) A mixture of flags in different banks of data memory can be specified in the operand fields.
In this case, the assembler automatically generates an instruction (SETBANK) for setting the bank number defined by an <expression> in the bank register, and generates an instruction for flag operation after bank switching.

(3) In the operand fields, a flag defined in data memory and/or a flag defined in the register file can be specified. The register file area (40H-7FH) in which data memory and addresses overlap each other is processed as a data memory area.

(4) If the number of operands is 0 or greater than 4, an error (F037: Syntax error) occurs. In this case, the line is invalidated and is not expanded, and an NOP instruction is generated.

(5) When a symbol is specified in an operand field, the symbol must be defined before the SETX instruction. If a symbol is not defined beforehand or is defined after being referenced, an error (F058: Undefined symbol) occurs, and the line is invalidated. Accordingly, the built-in macro is not expanded, and an NOP instruction is generated.

(6) If an external definition symbol is specified in an operand field, an error (F150: Impossible to write the external symbol) occurs, and the line is invalidated. Accordingly, the built-in macro is not expanded, and an NOP instruction is generated.

(7) Built-in macros allow nesting in the same way as for ordinary macros. This means that the nesting of built-in macros, ordinary macros, conditional assembly pseudo instructions, and repetitive pseudo instructions is allowed up to 40 levels. If the depth of nesting exceeds 41 levels, an abort error (A035: Nesting overflow) occurs, and assembly processing is terminated.

(8) If the bank number obtained when an <expression> is evaluated is not used with the product, an error (F046: Invalid BANK No.) occurs, and an NOP instruction is generated.

(9) If the data memory address obtained when an <expression> is evaluated is not used with the product, an error (F067: Address error) occurs, and an NOP instruction is generated.

(10) If the bit position value obtained when an <expression> is evaluated is a value other than 1, 2, 4, or 8, an error (F044: Invalid value) occurs, and an NOP instruction is generated.

[Sample expansion]

[Example 1] When one flag is specified

It is assumed that 0 is set in the bank register when the SETX instruction is executed. In this case, SETX is expanded to one instruction.

```
F1      FLG     0.01H.0

        SETX    F1
         ↓
        OR      .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
```

[Example 2] When four flags are specified, each located at a different address

It is assumed that 1 is set in the bank register when the SETX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     0.00H.0
F2      FLG     0.10H.1
F3      FLG     0.20H.2
F4      FLG     0.30H.3

        SETX    F1,F2,F3,F4
         ↓
        MOV     BANK,#.DF.(F1)SHR 12 AND 0FH
        OR      .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        OR      .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        OR      .MF.(F3)SHR 4,#.DF.(F3)AND 0FH
        OR      .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
```

[Example 3] When two flags are specified, each located at a different bank and address

It is assumed that 1 is set in the bank register when the SETX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     1.00H.0
F2      FLG     2.10H.1

        SETX    F1,F2
         ↓
        OR      .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        MOV     BANK,#.DF.(F2)SHR 12 AND 0FH
        OR      .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
```

21.3.2 CLRX

[Format]

```
[<label>:][Δ]CLRXΔ<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]CLRXΔ<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]CLRXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]CLRXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
```

[Function]

The flag specified in each operand field is reset to 0.

[Notes]

The notes relating to CLRX are the same as those for SETX. See **Section 21.3.1**.

[Sample expansion]

[Example 1] When one flag is specified

It is assumed that 0 is set in the bank register when the CLRX instruction is executed. In this case, CLRX is expanded to one instruction.

```
F1      FLG     0.01H.0

        CLRX    F1
         ↓
        AND     .MF.(F1)SHR 4,#.DF.(NOT(F1))AND 0FH
```

[Example 2] When four flags are specified, each located at a different address

It is assumed that 1 is set in the bank register when the CLRX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     0.00H.0
F2      FLG     0.10H.1
F3      FLG     0.20H.2
F4      FLG     0.30H.3

        CLRX    F1,F2,F3,F4
         ↓
        MOV     BANK,#.DF.(F1)SHR 12 AND 0FH
        AND     .MF.(F1)SHR 4,#.DF.(NOT(F1))AND 0FH
        AND     .MF.(F2)SHR 4,#.DF.(NOT(F2))AND 0FH
        AND     .MF.(F3)SHR 4,#.DF.(NOT(F3))AND 0FH
        AND     .MF.(F4)SHR 4,#.DF.(NOT(F4))AND 0FH
```

[Example 3] When two flags are specified, each located at a different bank and address

It is assumed that 1 is set in the bank register when the CLRX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     1.00H.0
F2      FLG     2.10H.1

        CLRX    F1,F2
         ↓
        AND     .MF.(F1)SHR 4,#.DF.(NOT(F1))AND 0FH
        MOV     BANK,#.DF.(F2)SHR 12 AND 0FH
        AND     .MF.(F2)SHR 4,#.DF.(NOT(F2))AND 0FH
```

21.3.3 NOTX

[Format]

```
[<label>:][Δ]NOTXΔ<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]NOTXΔ<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]NOTXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]NOTXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
```

[Function]

The flag specified in each operand field is inverted.

[Notes]

The notes relating to NOTX are the same as those for SETX. See **Section 21.3.1**.

[Sample expansion]

[Example 1] When one flag is specified

It is assumed that 0 is set in the bank register when the NOTX instruction is executed. In this case, NOTX is expanded to one instruction.

```
F1      FLG     0.01H.0

        NOTX    F1
         ↓
        XOR     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
```

[Example 2] When four flags are specified

It is assumed that 1 is set in the bank register when the NOTX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     0.00H.0
F2      FLG     0.10H.1
F3      FLG     0.20H.2
F4      FLG     0.30H.3

        NOTX    F1,F2,F3,F4
         ↓
        MOV     BANK,#.DF.(F1)SHR 12 AND 0FH
        XOR     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        XOR     .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        XOR     .MF.(F3)SHR 4,#.DF.(F3)AND 0FH
        XOR     .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
```

[Example 3] When two flags are specified, each located at a different bank and address

It is assumed that 1 is set in the bank register when the NOTX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     1.00H.0
F2      FLG     2.10H.1

        NOTX    F1,F2
         ↓
        XOR     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        MOV     BANK,#.DF.(F2)SHR 12 AND 0FH
        XOR     .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
```

21.3.4 SKTX

[Format]

```
[<label>:][Δ]SKTXΔ<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SKTXΔ<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SKTXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SKTXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
```

[Function]

When the flag specified in each operand field is set to 1, the next instruction is skipped.

[Notes]

The notes relating to SKTX are the same as those for SETX. See **Section 21.3.1**.

[Sample expansion]

[Example 1] When one flag is specified

It is assumed that 0 is set in the bank register when the SKTX instruction is executed. In this case, SKTX is expanded to one instruction.

```
F1      FLG     0.01H.0

        SKTX    F1
         ↓
        SKT     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
```

[Example 2] When four flags are specified, each located at a different address

It is assumed that 1 is set in the bank register when the SKTX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     0.00H.0
F2      FLG     0.10H.1
F3      FLG     0.20H.2
F4      FLG     0.30H.3

        SKTX    F1,F2,F3,F4
         ↓
        MOV     BANK,#.DF.(F1)SHR 12 AND 0FH
        SKT     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        BR      ?L0
        SKT     .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        BR      ?L0
        SKF     .MF.(F3)SHR 4,#.DF.(F3)AND 0FH
        SKT     .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
        ?L0:
```

[Example 3] When two flags are specified, each located at a different bank and address

It is assumed that 1 is set in the bank register when the SKTX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     1.00H.0
F2      FLG     2.10H.1

        SKTX    F1,F2
         ↓
        SKT     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        BR      ?L0
        MOV     BANK,#.DF.(F2)SHR 12 AND 0FH
        SKT     .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        ?L0:
```

21.3.5 SKFX

[Format]

```
[<label>:][Δ]SKFXΔ<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SKFXΔ<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SKFXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]SKFXΔ<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
```

[Function]

When the flag specified in each operand field is reset to 0, the next instruction is skipped.

[Notes]

The notes relating to SKFX are the same as those for SETX. See **Section 21.3.1**.

[Sample expansion]

[Example 1] When one flag is specified

It is assumed that 0 is set in the bank register when the SKFX instruction is executed. In this case, SKFX is expanded to one instruction.

```
F1      FLG     0.01H.0

        SKFX    F1
         ↓
        SKF     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
```

[Example 2] When four flags are specified

It is assumed that 1 is set in the bank register when the SKFX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     0.00H.0
F2      FLG     0.10H.1
F3      FLG     0.20H.2
F4      FLG     0.30H.3

        SKFX    F1,F2,F3,F4
         ↓
        MOV     BANK,#.DF.(F1)SHR 12 AND 0FH
        SKF     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        BR      ?L0
        SKF     .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        BR      ?L0
        SKT     .MF.(F3)SHR 4,#.DF.(F3)AND 0FH
        SKF     .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
        ?L0:
```

[Example 3] When two flags are specified, each located at a different bank and address

It is assumed that 1 is set in the bank register when the SKFX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     1.00H.0
F2      FLG     2.10H.1

        SKFX    F1,F2
         ↓
        SKF     .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        BR      ?L0
        MOV     BANK,#.DF.(F2)SHR 12 AND 0FH
        SKF     .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        ?L0:
```

21.3.6 INITFLGX

[Format]

```
[<label>:][Δ]INITFLGXΔ<[NOT|INV]FLG-type-expression-list>[Δ][;<comment>]
```

[Function]

The flag(s) specified in the operand field are initialized to 0 if the flag name(s) are preceded by NOT or INV; if the flag name(s) are preceded by neither NOT nor INV, the flag(s) are initialized to 1. Up to four flags can be specified in the operand field.

[Notes]

The notes relating to INITFLGX are the same as those for SETX. See **Section 21.3.1**.

[Sample expansion]

[Example 1] When one flag is specified

It is assumed that 0 is set in the bank register when the INITFLGX instruction is executed. In this case, INITFLGX is expanded to one instruction.

```
F1      FLG         0.01H.0

        INITFLGX    F1
         ↓
        OR          .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
```

[Example 2] When four flags are specified

It is assumed that 1 is set in the bank register when the INITFLGX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG     0.00H.0
F2      FLG     0.10H.1
F3      FLG     0.20H.2
F4      FLG     0.30H.3

        INITFLGX    F1,NOT F2,NOT F3,F4
         ↓
        MOV         BANK,#.DF.(F1)SHR 12 AND 0FH
        OR          .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        AND         .MF.(F2)SHR 4,#.DF.(NOT(F2))AND 0FH
        AND         .MF.(F3)SHR 4,#.DF.(NOT(F3))AND 0FH
        OR          .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
```

[Example 3] When two flags are specified, each located at a different bank and address

It is assumed that 1 is set in the bank register when the INITFLGX instruction is executed. In this case, a bank set instruction is expanded.

```
F1      FLG      1.00H.0
F2      FLG      2.10H.1

        INITFLGX     F1,NOT F2
         ↓
        OR           .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
        MOV          BANK,#.DF.(F2)SHR 12 AND 0FH
        AND          .MF.(F2)SHR 4,#.DF.(NOT(F2)) AND 0FH
```

21.3.7 MOVX

[Format]

```
[<label>:][Δ]MOVXΔM,#i[Δ][;<comment>]
[<label>:][Δ]MOVXΔ@M1,M2[Δ][;<comment>]
[<label>:][Δ]MOVXΔM1,@M2[Δ][;<comment>]
[<label>:][Δ]MOVXΔ<expression (FLG type)>,<expression (FLG type)>[Δ][;<comment>]
[<label>:][Δ]MOVXΔM1,M2[Δ][;<comment>]
```

[Function]

(1) MOVX M,#i

In the data memory location specified in M, immediate data i is stored.

(2) MOVX @M1,M2

- When MPE = 1

  `[(MP),(M1)] ← (M2)`

- When MPE = 0

  `[mH,(M1)] ← (M2)` mH: Row address of M2

Remarks 1. The data at the data memory location specified in M2 is stored into the data memory location specified in M1.

2. When MPE = 0, a transfer occurs within the same row address as M2.

(3) MOVX M1,@M2

- When MPE = 1

  `(M1) ← [(MP),(M2)]`

- When MPE = 0

  `(M1) ← [mH,(M2)]` mH: Row address of M1

Remarks 1. The data of the data memory location specified in M2 is stored into the data memory location specified in M1.

2. When MPE = 0, a transfer occurs within the same row address as M1.

(4) MOV M1,M2

The data of the memory location specified in M2 is transferred to the memory location specified in M1.

Remarks 1. The MEM-type symbol specified in each operand may have a different bank specified. In this case, the assembler generates an instruction for setting a bank for automatic bank switching.

2. Horizontal data can be transferred to a vertical data area, or vice versa. For example, four nibbles of horizontal data can be transferred to a vertical four-nibble area.

(5) MOVX <expression (FLG type)>,<expression (FLG type)>
The value of the flag specified in the second operand <expression (FLG type)> is stored into the flag specified in the first operand <expression (FLG type)>. This operation transfers only one bit.

Remark The FLG-type symbol specified in each operand may have a different bank specified. In this case, the assembler generates an instruction for setting a bank for automatic bank switching.

[Notes]

(1) In operand M, <expression (MEM type)> can be specified.

(2) When immediate data is specified, a 32-bit value can be specified. If a value of more than 32 bits is specified, the data beyond the 32 bits is ignored.

(3) When multiple nibbles are transferred with the MOVX @M,M or MOVX M,@M instruction, the value of the memory location specified in @M is incremented by the number of nibbles that have been transferred.

[Sample expansion]

[Example 1] Expansion of MOVX M,#i
Four-nibble immediate data is transferred to a 4-nibble data memory location.

```
M1      NIBBLE4          0.00H

        MOVX M1,#1234H
         ↓
        MOV       (M1)+3H,#(1234H)AND 0FH
        MOV       (M1)+2H,#(1234H)SHR 4 AND 0FH
        MOV       (M1)+1H,#(1234H)SHR 8 AND 0FH
        MOV       (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] Expansion of MOVX @M1,M2

The data of the data memory location specified in M1 is transferred to the data memory location specified in M2.

It is assumed that the bank register (BANK) and register pointer (RP) hold the same symbol value as that specified in M1.

```
M0      NIBBLE          0.08H
M1      NIBBLE4         0.00H

        MOVX      @M0,M1
         ↓
        ADD       (M0),#03H
        MOV       @(M0),(M1)+03H
        SUB       (M0),#01H
        MOV       @(M0),(M1)+02H
        SUB       (M0),#01H
        MOV       @(M0),(M1)+01H
        SUB       (M0),#01H
        MOV       @(M0),(M1)
```

[Example 3] Expansion of MOVX M1,@M2

The data of the data memory location specified in M2 is transferred to the data memory location specified in M1.

```
M0      NIBBLE    0.08H
M1      NIBBLE4   0.38H

        MOVX      M1,@M0
         ↓
        ADD       (M0),#03H
        MOV       (M1)+03H,@(M0)
        SUB       (M0),#01H
        MOV       (M1)+02H,@(M0)
        SUB       (M0),#01H
        MOV       (M1)+01H,@(M0)
        SUB       (M0),#01H
        MOV       (M1),@(M0)
```

[Example 4] Expansion of MOVX M,M

Horizontal 4-nibble data is transferred to a vertical 4-nibble data area.

It is assumed that 0 is set in the bank register, and that 0 is set in the register pointer.

```
M0      NIBBLE4V  0.01H
M1      NIBBLE4   0.38H

        MOVX      M0,M1
         ↓
        MOV       RPH,#.DM.(M0)SHR 8 AND 0FH
        AND       RPL,#01H
        OR        RPH,#.DM.((M0)+030H)SHR 3 AND 0EH
        LD        (M0),(M1)+03H
        AND       RPL,#01H
        OR        RPH,#.DM.((M0)+020H)SHR 3 AND 0EH
        LD        (M0),(M1)+02H
        AND       RPL,#01H
        OR        RPH,#.DM.((M0)+010H)SHR 3 AND 0EH
        LD        (M0),(M1)+01H
        AND       RPL,#01H
        OR        RPH,#.DM.(M0)SHR 3 AND 0EH
        LD        (M0),(M1)
```

[Example 5] Expansion 1 of MOVX <expression (FLG type)>,<expression (FLG type)>

Only a 1-bit value is transferred from one flag to another.

```
F1      FLG       0.10H.0
F2      FLG       0.20H.0

        MOVX      F1,F2
         ↓
        SKT       .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        AND       .MF.(F1)SHR 4,#.DF.NOT(F1)AND 0FH
        SKF       .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
        OR        .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
```

[Example 6] Expansion 2 of MOVX <expression (FLG type)>,<expression (FLG type)>
Only a one-bit value is transferred from the RF flag.

```
RF1     FLG     0.90H.0
RF2     FLG     0.0A5H.0

        MOVX    RF1,RF2
         ↓
        PEEK    WR,.MF.(RF2)SHR 4
        SKT     WR,#.DF.(RF2)AND 0FH
        BR      ?L0
        PEEK    WR,.MF.(RF1)SHR 4
        OR      WR,#.DF.(RF1)AND 0FH
        BR      ?L1
        ?L0:
        PEEK    WR,.MF.(RF1)SHR 4
        AND     WR,#.DF.NOT(RF1)AND 0FH
        ?L1:
        POKE    .MF.(RF1)SHR 4,WR
```

21.3.8 MOVTX

[Format]

```
[<label>:][Δ]MOVTXΔ<expression>[Δ][;<comment>]
```

[Function]

The value of the program memory location specified in the AR register is stored into data buffer DBF. At this time, the EPA bit is determined from <table-name> specified in the operand field.

[Notes]

As <expression> in the operand field, only a label-type symbol can be specified. If a symbol of a type other than the label type is specified, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated, and an NOP instruction is generated.

[Sample expansion]

```
MOVTX          TBL1
 ↓
MOV      AR3,#.DL.(TBL1)SHR 12 AND 0FH
MOV      AR2,#.DL.(TBL1)SHR 8 AND 0FH
MOV      AR1,#.DL.(TBL1)SHR 4 AND 0FH
MOV      AR0,#.DL.(TBL1)AND 0FH
MOVT     DBF,@AR
```

21.3.9 ADDX

[Format]

```
[<label>:][Δ]ADDXΔM,#i[Δ][;<comment>]
[<label>:][Δ]ADDXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) + i

Immediate data i is added to the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

(2) M1 ← (M1) + (M2)

The value of the data memory location specified in the second operand is added to the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

- When CMP = 1

(1) (M) + i

The result of addition is not stored, but only the flag changes.

(2) (M1) + (M2)

The result of addition is not stored, but only the flag changes.

[Operation-based flag manipulation]

(1) If a carry is generated in an addition, the carry flag (CY) is set. If no carry is generated in an addition, the carry flag (CY) is reset.

(2) If an addition produces a value other than 0, the zero flag (Z) is reset.

(3) If an addition produces a zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of addition can be performed: binary addition and BCD addition. The type of addition to be performed can be specified using the BCD flag (BCD) of PSW.

[Notes]

(1) In the first operand, only <expression (MEM type)> can be specified. If the type obtained when the expression is evaluated is other than MEM, an error (F011: Illegal first operand type) occurs. In this case, the extended instruction is not expanded, and an NOP instruction is generated.

(2) In the second operand, only immediate data using # or <expression (MEM type)> can be specified. If any other data or value is specified, an error (F012: Illegal second operand type) occurs. In this case, the extended instruction is not expanded, and an NOP instruction is generated.

[Sample expansion]

[Example 1] Expansion of ADDX M,#i

Four-nibble immediate data is added to a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4     0.00H

        ADDX    M1,#1234H
         ↓
        ADD     (M1)+03H,#(1234H)AND 0FH
        ADDC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        ADDC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        ADDC    (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] Expansion of ADDX M,M

The value of a 4-nibble data memory location is added to the value of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4     0.00H
M5      NIBBLE4     0.38H

        ADDX    M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        ADD     (M1)+03H,(M5)+03H
        ADDC    (M1)+02H,(M5)+02H
        ADDC    (M1)+01H,(M5)+01H
        ADDC    (M1),(M5)
```

21.3.10 ADDCX

[Format]

```
[<label>:][Δ]ADDCXΔM,#i[Δ][;<comment>]
[<label>:][Δ]ADDCXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) + i + CY
Immediate data i and CY are added to the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

(2) M1 ← (M1) + (M2) + CY
The value of the data memory location specified in the second operand and CY are added to the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

- When CMP = 1

(1) (M) + i + CY
The result of addition is not stored, but only the flag changes.

(2) (M1) + (M2) + CY
The result of addition is not stored, but only the flag changes.

[Operation-based flag manipulation]

(1) If a carry is generated in an addition, the carry flag (CY) is set. If no carry is generated in an addition, the carry flag (CY) is reset.

(2) If an addition produces a value other than 0, the zero flag (Z) is reset.

(3) If an addition produces a zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of addition can be performed: binary addition and BCD addition. The type of addition to be performed is specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to ADDCX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Expansion of ADDCX M,#i

Four-nibble immediate data is added to a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        ADDCX   M1,#1234H
         ↓
        ADDC    (M1)+03H,#(1234H)AND 0FH
        ADDC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        ADDC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        ADDC    (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] Expansion of ADDCX M,M

The value of a 4-nibble data memory location is added to the value of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        ADDCX   M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        ADDC    (M1)+03H,(M5)+03H
        ADDC    (M1)+02H,(M5)+02H
        ADDC    (M1)+01H,(M5)+01H
        ADDC    (M1),(M5)
```

21.3.11 ADDSX

[Format]

```
[<label>:][Δ]ADDSXΔM,#i[Δ][;<comment>]
[<label>:][Δ]ADDSXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) + i (Skip if CY = 1)
Immediate data i is added to the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a carry is generated, the next instruction is skipped.

(2) M1 ← (M1) + (M2) (Skip if CY = 1)
The value of the data memory location specified in the second operand is added to the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a carry is generated, the next instruction is skipped.

- When CMP = 1

(1) (M) + i (Skip if CY = 1)
The result of addition is not stored, so only the flag changes.

(2) (M1) + (M2) (Skip if CY = 1)
The result of addition is not stored, so only the flag changes.

[Operation-based flag manipulation]

(1) If a carry is generated in an addition, the carry flag (CY) is set. If no carry is generated in an addition, the carry flag (CY) is reset.

(2) If an addition produces a value other than 0, the zero flag (Z) is reset.

(3) If an addition produces a zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of addition can be performed: binary addition and BCD addition. The type of addition to be performed is specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to ADDSX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Expansion of ADDSX M,#i

Four-nibble immediate data is added to a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        ADDSX   M1,#1234H
         ↓
        ADD     (M1)+03H,#(1234H)AND 0FH
        ADDC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        ADDC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        ADDC    (M1),#(1234H)SHR 12 AND 0FH
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

[Example 2] Expansion of ADDSX M,M

The value of a 4-nibble data memory location is added to the value of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        ADDSX   M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPH,#.DM.(M1)SHR 3 AND 0EH
        ADD     (M1)+03H,(M5)+03H
        ADDC    (M1)+02H,(M5)+02H
        ADDC    (M1)+01H,(M5)+01H
        ADDC    (M1),(M5)
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

21.3.12 ADDCSX

[Format]

```
[<label>:][Δ]ADDCSXΔM,#i[Δ][;<comment>]
[<label>:][Δ]ADDCSXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) + i + CY (Skip if CY = 1)
Immediate data i and CY are added to the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a carry is generated, the next instruction is skipped.

(2) M1 ← (M1) + (M2) + CY (Skip if CY = 1)
The value of the data memory location specified in the second operand and CY are added to the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a carry is generated, the next instruction is skipped.

- When CMP = 1

(1) (M) + i + CY (Skip if CY = 1)
The result of addition is not stored, so only the flag changes.

(2) (M1) + (M2) + CY (Skip if CY = 1)
The result of addition is not stored, so only the flag changes.

[Operation-based flag manipulation]

(1) If a carry is generated by an addition, the carry flag (CY) is set. If no carry is generated by an addition, the carry flag (CY) is reset.

(2) If addition results in a value other than 0, the zero flag (Z) is reset.

(3) If addition results in zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of addition can be performed: binary addition and BCD addition. The type of addition to be performed can be specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to ADDCSX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Four-nibble immediate data is added to a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        ADDCSX    M1,#1234H
         ↓
        ADDC    (M1)+03H,#(1234H)AND 0FH
        ADDC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        ADDC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        ADDC    (M1),#(1234H)SHR 12 AND 0FH
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

[Example 2] The value of a 4-nibble data memory location is added to that of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        ADDCSX    M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPH,#.DM.(M1)SHR 3 AND 0EH
        ADDC    (M1)+03H,(M5)+3H
        ADDC    (M1)+02H,(M5)+2H
        ADDC    (M1)+01H,(M5)+1H
        ADDC    (M1),(M5)
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

21.3.13 SUBX

[Format]

```
[<label>:][Δ]SUBXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SUBXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) - i

Immediate data i is subtracted from the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

(2) M1 ← (M1) - (M2)

The value of the data memory location specified in the second operand is subtracted from the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

- When CMP = 1

(1) (M) - i

The result of subtraction is not stored, so only the flag changes.

(2) (M1) - (M2)

The result of subtraction is not stored, so only the flag changes.

[Operation-based flag manipulation]

(1) If a borrow is generated by a subtraction, the carry flag (CY) is set. If no borrow is generated by a subtraction, the carry flag (CY) is reset.

(2) If subtraction results in a value other than 0, the zero flag (Z) is reset.

(3) If subtraction results in zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of subtraction can be performed: binary subtraction and BCD subtraction. The type of subtraction to be performed can be specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to SUBX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Four-nibble immediate data is subtracted from a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SUBX      M1,#1234H
         ↓
        SUB     (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] The value of a 4-nibble data memory location is subtracted from that of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SUBX      M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPH,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+3H
        SUBC    (M1)+02H,(M5)+2H
        SUBC    (M1)+01H,(M5)+1H
        SUBC    (M1),(M5)
```

21.3.14 SUBCX

[Format]

```
[<label>:][Δ]SUBCXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SUBCXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) - i - CY

Immediate data i and CY are subtracted from the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

(2) M1 ← (M1) - (M2) - CY

The value of the data memory location specified in the second operand and CY are subtracted from the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand.

- When CMP = 1

(1) (M) - i - CY

The result of subtraction is not stored, so only the flag changes.

(2) (M1) - (M2) - CY

The result of subtraction is not stored, so only the flag changes.

[Operation-based flag manipulation]

(1) If a borrow is generated by a subtraction, the carry flag (CY) is set. If no borrow is generated by a subtraction, the carry flag (CY) is reset.

(2) If subtraction results in a value other than 0, the zero flag (Z) is reset.

(3) If subtraction results in zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of subtraction can be performed: binary subtraction and BCD subtraction. The type of subtraction to be performed can be specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to SUBCX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Four-nibble immediate data is subtracted from a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SUBCX     M1,#1234H
         ↓
        SUBC    (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] The value of a 4-nibble data memory location is subtracted from that of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SUBCX     M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUBC    (M1)+03H,(M5)+3H
        SUBC    (M1)+02H,(M5)+2H
        SUBC    (M1)+01H,(M5)+1H
        SUBC    (M1),(M5)
```

21.3.15 SUBSX

[Format]

```
[<label>:][Δ]SUBSXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SUBSXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) - i (Skip if CY = 1)
Immediate data i is subtracted from the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a borrow is generated, the next instruction is skipped.

(2) M1 ← (M1) - (M2) (Skip if CY = 1)
The value of the data memory location specified in the second operand is subtracted from the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a borrow is generated, the next instruction is skipped.

- When CMP = 1

(1) (M) - i (Skip if CY = 1)
The result of subtraction is not stored, so only the flag changes.

(2) (M1) - (M2) (Skip if CY = 1)
The result of subtraction is not stored, so only the flag changes.

[Operation-based flag manipulation]

(1) If a borrow is generated by a subtraction, the carry flag (CY) is set. If no borrow is generated by a subtraction, the carry flag (CY) is reset.

(2) If subtraction results in a value other than 0, the zero flag (Z) is reset.

(3) If subtraction results in zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of subtraction can be performed: binary subtraction and BCD subtraction. The type of subtraction to be performed can be specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to SUBSX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Four-nibble immediate data is subtracted from a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SUBSX     M1,#1234H
         ↓
        SUB     (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

[Example 2] The value of a 4-nibble data memory location is subtracted from that of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SUBSX     M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+3H
        SUBC    (M1)+02H,(M5)+2H
        SUBC    (M1)+01H,(M5)+1H
        SUBC    (M1),(M5)
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

21.3.16 SUBCSX

[Format]

```
[<label>:][Δ]SUBCSXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SUBCSXΔM1,M2[Δ][;<comment>]
```

[Function]

- When CMP = 0

(1) M ← (M) - i - CY (Skip if CY = 1)
Immediate data i and CY are subtracted from the value of the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a borrow is generated, the next instruction is skipped.

(2) M1 ← (M1) - (M2) - CY (Skip if CY = 1)
The value of the data memory location specified in the second operand and CY are subtracted from the data memory location specified in the first operand, then the result is stored into the data memory location specified in the first operand. If a borrow is generated, the next instruction is skipped.

- When CMP = 1

(1) (M) - i - CY (Skip if CY = 1)
The result of subtraction is not stored, so only the flag changes.

(2) (M1) - (M2) - CY (Skip if CY = 1)
The result of subtraction is not stored, so only the flag changes.

[Operation-based flag manipulation]

(1) If a borrow is generated by a subtraction, the carry flag (CY) is set. If no borrow is generated by a subtraction, the carry flag (CY) is reset.

(2) If subtraction results in a value other than 0, the zero flag (Z) is reset.

(3) If subtraction results in zero, the zero flag (Z) is set when the compare flag is reset (CMP = 0); the zero flag (Z) does not change when the compare flag is set (CMP = 1).

(4) Two types of subtraction can be performed: binary subtraction and BCD subtraction. The type of subtraction to be performed can be specified using the BCD flag (BCD) of PSW.

[Notes]

The notes relating to SUBCSX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] Four-nibble immediate data is subtracted from a 4-nibble data memory location (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SUBCSX    M1,#1234H
         ↓
        SUBC    (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

[Example 2] The value of a 4-nibble data memory location is subtracted from that of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SUBCSX    M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUBC    (M1)+03H,(M5)+3H
        SUBC    (M1)+02H,(M5)+2H
        SUBC    (M1)+01H,(M5)+1H
        SUBC    (M1),(M5)
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

21.3.17 SKEX

[Format]

```
[<label>:][Δ]SKEXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SKEXΔM1,M2[Δ][;<comment>]
```

[Function]

(1) SKEX M,#i

When the value of the data memory location specified in the first operand is equal to that of immediate data i, specified in the second operand, the next instruction is skipped.

(2) SKEX M1,M2

When the value of the data memory location specified in the first operand is equal to that of the data memory location specified in the second operand, the next instruction is skipped.

[Notes]

The notes relating to SKEX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is compared with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SKEX      M1,#1234H
         ↓
        SKNE    (M1)+03H,#(1234H)AND 0FH
        SKE     (M1)+02H,#(1234H)SHR 4 AND 0FH
        BR      ?L0
        SKNE    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SKE     (M1),#(1234H)SHR 12 AND 0FH
        ?L0:
```

[Example 2] The value of a 6-nibble data memory location is compared with 6-nibble immediate data (with 0 set in the bank register).

This expansion differs from an expansion performed with 4-nibble operands.

```
M1      NIBBLE6   0.00H

        SKEX      M1,#123456H
         ↓
        OR      .MF.CY SHR 4,#.DF.(CMP OR Z)AND 0FH
        SUB     (M1)+05H,#(123456H)AND 0FH
        SUBC    (M1)+04H,#(123456H)SHR 4 AND 0FH
        SUBC    (M1)+03H,#(123456H)SHR 8 AND 0FH
        SUBC    (M1)+02H,#(123456H)SHR 12 AND 0FH
        SUBC    (M1)+01H,#(123456H)SHR 16 AND 0FH
        SUBC    (M1),#(123456H)SHR 20 AND 0FH
        SKT     .MF.Z SHR 4,#.DF.Z AND 0FH
```

[Example 3] The value of a 4-nibble data memory location is compared with that of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SKEX      M1,M5
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z) AND 0FH
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0FH
        SUB     (M1)+03H,(M5)+3H
        SUBC    (M1)+02H,(M5)+2H
        SUBC    (M1)+01H,(M5)+1H
        SUBC    (M1),(M5)
        SKT     .MF.Z SHR 4,#.DF.Z AND 0FH
```

21.3.18 SKNEX

[Format]

```
[<label>:][Δ]SKNEX Δ M,#i[Δ][;<comment>]
[<label>:][Δ]SKNEX Δ M1,M2[Δ][;<comment>]
```

[Function]

(1) SKNEX M,#i

When the value of the data memory location specified in the first operand is not equal to that of immediate data i, specified in the second operand, the next instruction is skipped.

(2) SKNEX M1,M2

When the value of the data memory location specified in the first operand is not equal to that of the data memory location specified in the second operand, the next instruction is skipped.

[Notes]

The notes relating to SKNEX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is compared with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SKNEX     M1,#1234H
         ↓
        SKE     (M1)+03H,#(1234H)AND 0FH
        BR      $+6
        SKNE    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SKE     (M1)+01H,#(1234H)SHR 8 AND 0FH
        BR      $+3
        SKNE    (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] The value of a 6-nibble data memory location is compared with 6-nibble immediate data (with 0 set in the bank register).

This expansion differs from that performed with 4-nibble operands.

```
M1      NIBBLE6   0.00H

        SKNEX     M1,#123456H
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
        SUB     (M1)+05H,#(123456H)AND 0FH
        SUBC    (M1)+04H,#(123456H)SHR 4 AND 0FH
        SUBC    (M1)+03H,#(123456H)SHR 8 AND 0FH
        SUBC    (M1)+02H,#(123456H)SHR 12 AND 0FH
        SUBC    (M1)+01H,#(123456H)SHR 16 AND 0FH
        SUBC    (M1),#(123456H)SHR 20 AND 0FH
        SKF     .MF.Z SHR 4,#.DF.Z AND 0FH
```

[Example 3] The value of a 4-nibble data memory location is compared with the value of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SKNEX     M1,M5
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z) AND 0FH
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+3H
        SUBC    (M1)+02H,(M5)+2H
        SUBC    (M1)+01H,(M5)+1H
        SUBC    (M1),(M5)
        SKF     .MF.Z SHR 4,#.DF.Z AND 0FH
```

21.3.19 SKGEX

[Format]

```
[<label>:][Δ]SKGEXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SKGEXΔM1,M2[Δ][;<comment>]
```

[Function]

(1) SKGEX M,#i

When the value of the data memory location specified in the first operand is greater than or equal to that of immediate data i specified in the second operand, the next instruction is skipped.

(2) SKGEX M1,M2

When the value of the data memory location specified in the first operand is greater than or equal to that of the data memory location specified in the second operand, the next instruction is skipped.

[Notes]

The notes relating to SKGEX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is compared with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SKGEX     M1,#1234H
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
        SUB     (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
        SKF     .MF.CY SHR 4,#.DF.CY AND 0FH
```

[Example 2] The value of a 4-nibble data memory location is compared with the value of another 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SKGEX     M1,M5
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z) AND 0FH
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+03H
        SUBC    (M1)+02H,(M5)+02H
        SUBC    (M1)+01H,(M5)+01H
        SUBC    (M1),(M5)
        SKF     .MF.CY SHR 4,#.DF.CY AND 0FH
```

21.3.20 SKGTX

[Format]

```
[<label>:][Δ]SKGTXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SKGTXΔM,M[Δ][;<comment>]
```

[Function]

(1) SKGTX M,#i

When the value of the data memory location specified in the first operand is greater than that of immediate data i specified in the second operand, the next instruction is skipped.

(2) SKGTX M,M

When the value of the data memory location specified in the first operand is greater than that of the data memory location specified in the second operand, the next instruction is skipped.

[Notes]

The notes relating to SKGTX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is compared with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SKGTX     M1,#1234H
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
        SUB     (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
        SKF     .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
```

[Example 2] The value of a 4-nibble general-purpose register is compared with that of a 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SKGTX     M1,M5
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z) AND 0FH
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+03H
        SUBC    (M1)+02H,(M5)+02H
        SUBC    (M1)+01H,(M5)+01H
        SUBC    (M1),(M5)
        SKF     .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
```

21.3.21 SKLEX

[Format]

```
[<label>:][Δ]SKLEXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SKLEXΔM,M[Δ][;<comment>]
```

[Function]

(1) SKLEX M,#i

When the value of the data memory location specified in the first operand is less than or equal to the value of immediate data i specified in the second operand, the next one instruction is skipped.

(2) SKLEX M,M

When the value of the data memory location specified in the first operand is less than or equal to that of the data memory location specified in the second operand, the next instruction is skipped.

[Notes]

The notes relating to SKLEX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is compared with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SKLEX     M1,#1234H
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
        SUB     (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
        SKF     .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
        SKT     (M1),#00H
```

[Example 2] The value of a 4-nibble general-purpose register is compared with that of a 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SKLEX     M1,M5
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z) AND 0FH
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+03H
        SUBC    (M1)+02H,(M5)+02H
        SUBC    (M1)+01H,(M5)+01H
        SUBC    (M1),(M5)
        SKF     .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
        SKT     (M1),#00H
```

21.3.22 SKLTX

[Format]

```
[<label>:][Δ]SKLTXΔM,#i[Δ][;<comment>]
[<label>:][Δ]SKLTXΔM,M[Δ][;<comment>]
```

[Function]

(1) SKLTX M,#i

When the value of the data memory location specified in the first operand is less than that of immediate data i, specified in the second operand, the next instruction is skipped.

(2) SKLTX M,M

When the value of the data memory location specified in the first operand is less than that of the data memory location specified in the second operand, the next instruction is skipped.

[Notes]

The notes relating to SKLTX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is compared with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        SKLTX     M1,#1234H
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
        SUB     (M1)+03H,#(1234H)AND 0FH
        SUBC    (M1)+02H,#(1234H)SHR 4 AND 0FH
        SUBC    (M1)+01H,#(1234H)SHR 8 AND 0FH
        SUBC    (M1),#(1234H)SHR 12 AND 0FH
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

[Example 2] The value of a 4-nibble general-purpose register is compared with that of a 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        SKLTX     M1,M5
         ↓
        OR      .MF.CMP SHR 4,#.DF.(CMP OR Z) AND 0FH
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        SUB     (M1)+03H,(M5)+03H
        SUBC    (M1)+02H,(M5)+02H
        SUBC    (M1)+01H,(M5)+01H
        SUBC    (M1),(M5)
        SKT     .MF.CY SHR 4,#.DF.CY AND 0FH
```

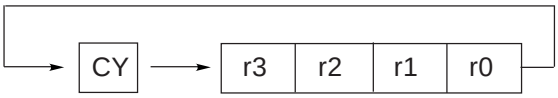
21.3.23 RORCX

[Format]

```
[<label>:][Δ]RORCXΔR[Δ][;<comment>]
```

[Function]

The contents of the general-purpose register specified in R and the carry flag are shifted one bit to the right.



When r MEM 0.00H

r0: bit0
r1: bit1
r2: bit2
r3: bit3

[Notes]

If an expression of a type other than <expression (MEM type)> is specified in operand R, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated, and an NOP instruction is generated.

[Sample expansion]

The contents of a 4-nibble general-purpose register are shifted one bit to the right.

```
R1        NIBBLE4     0.00H

          RORCX       R1
           ↓
          RORC    (R1)
          RORC    (R1)+01H
          RORC    (R1)+02H
          RORC    (R1)+03H
```

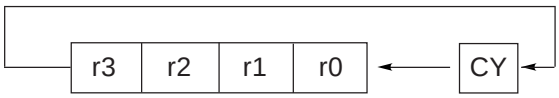
21.3.24 ROLCX

[Format]

```
[<label>:][Δ]ROLCXΔR[Δ][;<comment>]
```

[Function]

The contents of the general-purpose register specified in R and the carry flag are shifted one bit to the left.



When r MEM 0.00H
r0: bit0
r1: bit1
r2: bit2
r3: bit3

[Notes]

If an expression of a type other than <expression (MEM type)> is specified in operand R, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated, and an NOP instruction is generated.

[Sample expansion]

The contents of a 4-nibble general-purpose register are shifted one bit to the left.

```
R1        NIBBLE4     0.00H

          ROLCX       R1
           ↓
          ADDC    (R1)+03H,(R1)+03H
          ADDC    (R1)+02H,(R1)+02H
          ADDC    (R1)+01H,(R1)+01H
          ADDC    (R1),(R1)
```

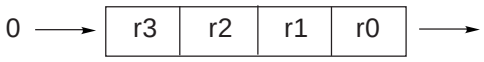
21.3.25 SHRX

[Format]

```
[<label>:][Δ]SHRXΔR[Δ][;<comment>]
```

[Function]

The contents of the general-purpose register specified in R are shifted one bit to the right.



When r MEM 0.00H

r0: bit0
r1: bit1
r2: bit2
r3: bit3

[Notes]

If an expression of a type other than <expression (MEM type)> is specified in operand R, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated, and an NOP instruction is generated.

[Sample expansion]

The contents of a 4-nibble general-purpose register are shifted one bit to the right.

```
R1      NIBBLE4     0.00H

        SHRX        R1
         ↓
        AND     .MF.CY SHR 4,#1011B
        RORC    (R1)
        RORC    (R1)+01H
        RORC    (R1)+02H
        RORC    (R1)+03H
```

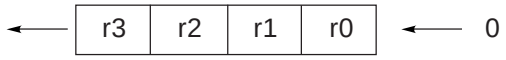
21.3.26 SHLX

[Format]

```
[<label>:][Δ]SHLXΔR[Δ][;<comment>]
```

[Function]

The contents of the general-purpose register specified in R are shifted one bit to the left.



When r MEM 0.00H

r0: bit0
r1: bit1
r2: bit2
r3: bit3

[Notes]

If an expression of a type other than <expression (MEM type)> is specified in operand R, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated, and an NOP instruction is generated.

[Sample expansion]

The contents of a 4-nibble general-purpose register are shifted one bit to the left.

```
R1          NIBBLE4     0.00H

            SHLX        R1
             ↓
            ADD     (R1)+03H,(R1)+03H
            ADDC    (R1)+02H,(R1)+02H
            ADDC    (R1)+01H,(R1)+01H
            ADDC    (R1),(R1)
```

21.3.27 ANDX

[Format]

```
[<label>:][Δ]ANDXΔM,#i[Δ][;<comment>]
[<label>:][Δ]ANDXΔM1,M2[Δ][;<comment>]
```

[Function]

(1) ANDX M,#i

M ← (M) AND i

The value of the data memory location specified in M is ANDed with immediate data i, then the result is stored into the data memory location specified in M.

(2) ANDX M1,M2

M1 ← (M1) AND (M2)

The value of the data memory location specified in M1 is ANDed with the value of the data memory location specified in M2, then the result is stored into the data memory location specified in M1.

[Notes]

The notes relating to ANDX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is ANDed with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        ANDX      M1,#1234H
         ↓
        AND     (M1)+03H,#(1234H)AND 0FH
        AND     (M1)+02H,#(1234H)SHR 4 AND 0FH
        AND     (M1)+01H,#(1234H)SHR 8 AND 0FH
        AND     (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] The value of a 4-nibble general-purpose register is ANDed with the value of a 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        ANDX      M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        AND     (M1)+03H,(M5)+03H
        AND     (M1)+02H,(M5)+02H
        AND     (M1)+01H,(M5)+01H
        AND     (M1),(M5)
```

21.3.28 ORX

[Format]

```
[<label>:][Δ]ORXΔM,#i[Δ][;<comment>]
[<label>:][Δ]ORXΔM1,M2[Δ][;<comment>]
```

[Function]

(1) ORX M,#i

M ← (M) OR i

The value of the data memory location specified in M is ORed with immediate data i, then the result is stored into the data memory location specified in M.

(2) ORX M1,M2

M1 ← (M1) OR (M2)

The value of the data memory location specified in M1 is ORed with the value of the data memory location specified in M2, then the result is stored into the data memory location specified in M1.

[Notes]

The notes relating to ORX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is ORed with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        ORX       M1,#1234H
         ↓
        OR      (M1)+03H,#(1234H)AND 0FH
        OR      (M1)+02H,#(1234H)SHR 4 AND 0FH
        OR      (M1)+01H,#(1234H)SHR 8 AND 0FH
        OR      (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] The value of a 4-nibble general-purpose register is ORed with the value of a 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        ORX       M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        OR      (M1)+03H,(M5)+03H
        OR      (M1)+02H,(M5)+02H
        OR      (M1)+01H,(M5)+01H
        OR      (M1),(M5)
```

21.3.29 XORX

[Format]

```
[<label>:][Δ]XORXΔM,#i[Δ][;<comment>]
[<label>:][Δ]XORXΔM1,M2[Δ][;<comment>]
```

[Function]

(1) XORX M,#i

M ← (M) XOR i

The value of the data memory location specified in M is exclusive-ORed with immediate data i, then the result is stored into the data memory location specified in M.

(2) XORX M1,M2

M1 ← (M1) OR (M2)

The value of the data memory location specified in M1 is exclusive-ORed with the value of the data memory location specified in M2, then the result is stored into the data memory location specified in M1.

[Notes]

The notes relating to XORX are the same as those for ADDX. See **Section 21.3.9**.

[Sample expansion]

[Example 1] The value of a 4-nibble data memory location is exclusive-ORed with 4-nibble immediate data (with 0 set in the bank register).

```
M1      NIBBLE4   0.00H

        XORX      M1,#1234H
         ↓
        XOR     (M1)+03H,#(1234H)AND 0FH
        XOR     (M1)+02H,#(1234H)SHR 4 AND 0FH
        XOR     (M1)+01H,#(1234H)SHR 8 AND 0FH
        XOR     (M1),#(1234H)SHR 12 AND 0FH
```

[Example 2] The value of a 4-nibble general-purpose register is exclusive-ORed with the value of a 4-nibble data memory location (with 0 set in the bank register, and 1 set in the register pointer).

```
M1      NIBBLE4   0.00H
M5      NIBBLE4   0.38H

        XORX      M1,M5
         ↓
        MOV     RPH,#.DM.(M1)SHR 8 AND 0FH
        AND     RPL,#01H
        OR      RPL,#.DM.(M1)SHR 3 AND 0EH
        XOR     (M1)+03H,(M5)+03H
        XOR     (M1)+02H,(M5)+02H
        XOR     (M1)+01H,(M5)+01H
        XOR     (M1),(M5)
```

21.3.30 BRX

[Format]

```
[<label>:][Δ]BRXΔ<expression>[Δ][;<comment>]
```

[Function]

BRX branches to the address specified in the operand field.
BRX is expanded to a direct branch instruction (BR <expression>).

[Notes]

(1) Only a label-type symbol can be specified for the <expression> in the operand field. If other than a label-type symbol is specified, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated and an NOP instruction is generated.

(2) If the address specified for the <expression> in the operand field is allocated in another segment, and a branch operation cannot be performed with a direct branch instruction at link time, the linker generates an indirect branch table in a free area.

[Sample expansion]

```
BRX       BR_LAB1
 ↓
BR        (BR_LAB1)
```

21.3.31 CALLX

[Format]

```
[<label>:][Δ]CALLXΔ<expression>[Δ][;<comment>]
```

[Function]

CALLX calls the subroutine located at the address specified in the operand field.
CALLX is expanded to a direct subroutine call instruction (CALL <expression>).

[Notes]

(1) Only a label-type symbol can be specified for the <expression> in the operand field. If other than a label-type symbol is specified, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated and an NOP instruction is generated.

(2) If the address specified for the <expression> in the operand field is allocated in another segment, and a call operation cannot be performed with a direct subroutine call instruction at link time, the linker generates an indirect subroutine call table in a free area.

[Sample expansion]

```
CALLX    CAL_LAB1
 ↓
CALLX    (CAL_LAB1)
```

21.3.32 SYSCALX

[Format]

```
[<label>:][Δ]SYSCALXΔ<expression (LAB type)>[Δ][;<comment>]
```

[Function]

SYSCALX calls the system subroutine located at the address specified in the operand field.

[Notes]

(1) Only a label-type symbol can be specified for the <expression (LAB type)> in the operand field. If other than a label-type symbol is specified, an error (F011: Illegal first operand type) occurs. In this case, the line is invalidated and one NOP instruction is generated as the object.

[Sample expansion]

```
EXTRN    LAB:EXTLAB
SYSCALX  EXTLAB
 ↓
SYSCALX    (((.DL.( EXTLAB )) SHR 4 ) AND 00F0H ) OR ((.DL.( EXTLAB )) AND 0FH)
```

* 21.4 STRUCTURED INSTRUCTIONS

Structured instructions are used for creating structured program descriptions.

The use of structured instructions improves the ease of programming as well as readability of the finished code. Accordingly, efficiency in debugging is also improved.

The following built-in macro instructions are provided to facilitate structured programming:

_IF ... _ELSEIF ... _ELSE ... _ENDIF
_WHILE ... _ENDW
_SWITCH ... _CASE ... _DEFAULT ... _ENDS
_REPEAT ... _UNTIL
_FOR ... _NEXT
_BREAK
_CONTINUE
_GOTO

These built-in macro instructions automatically generate branch instructions and branch destination labels based on condition checking.

[Relational operators]

The relational operators that can be used with structured instructions are shown below.

EQ (Equal) operator (==)
NE (Not Equal) operator (!=)
GE (Greater than or Equal) operator (>=)
GT (Greater than) operator (>)
LE (Less than or Equal) operator (<=)
LT (Less than) operator (<)

[Assignment operators]

The assignment operators that can be used with structured instructions are shown below.

- Simple assignment (=)
  [Format] $\alpha = \beta$
  [Function] Assigns β to α.
  [Explanation] α and β are assumed to be specifiable in extended instruction MOVTX which is a generation instruction.

- Addition assignment (+=)
  [Format] α += β
  [Function] Adds α and β, and assigns the result to α.
  [Explanation] α and β are assumed to be specifiable in extended instruction ADDX which is a generation instruction.

- Subtraction assignment (-=)
  [Format] α -= β
  [Function] Performs subtraction (α - β), and assigns the result to α.
  [Explanation] α and β are assumed to be specifiable in extended instruction SUBX, which is a generation instruction.

- Multiplication assignment (*=)
  [Format] α *= β
  [Function] Multiplies α and β (α * β), and assigns the result to α.
  [Explanation] α is assumed to be specifiable in extended instruction SHLX which is a generation instruction. For β, only immediate data is assumed to be specifiable. The value must be 2^n.

- Division assignment (/=)
  [Format] α /= β
  [Function] Performs bitwise division of α and β ($\alpha \div \beta$), and assigns the result to α.
  [Explanation] α is assumed to be specifiable in extended instruction SHRX which is a generation instruction. For β, only immediate data is assumed to be specifiable. Its value must be 2^n.

- AND assignment (&=)
  [Format] α &= β
  [Function] Performs bitwise AND for α and β (α & β), and assigns the result to α.
  [Explanation] α and β are assumed to be specifiable in extended instruction ANDX which is a generation instruction.

- OR assignment (|=)
  [Format] α |= β
  [Function] Performs bitwise OR for α and β (α | β), and assigns the result to α.
  [Explanation] α and β are assumed to be specifiable in extended ORX which is a generation instruction.

- Exclusive OR assignment (^=)
  [Format] α ^= β
  [Function] Performs bitwise exclusive OR for α and β (α ^ β), and assigns the result to α.
  [Explanation] α and β are assumed to be specifiable in extended instruction ORX which is a generation instruction.

- Shift-right assignment (>>=)
  [Format] α >>= β
  [Function] Shifts α right by β, and assigns the result to α.
  [Explanation] α is assumed to be specifiable in extended instruction SHRX which is a generation instruction. For β, only immediate data is assumed to be specifiable.

- Shift-left assignment (<<=)
  [Format] α <<= β
  [Function] Shifts α left by β, and assigns the result to α.
  [Explanation] α is assumed to be specifiable in extended instruction SHLX which is a generation instruction. For β, only immediate data is assumed to be specifiable.

- Increment (++)
  [Format] α ++
  [Function] Increments the content of α by one.
  [Explanation] A MEM-type symbol can be specified for α.

- Decrement (—)
  [Format] α —
  [Function] Decrements the content of α by one.
  [Explanation] A MEM-type symbol can be specified for α.

21.4.1 _IF ... _ELSEIF ... _ELSE ... _ENDIF

[Format]

```
_IF[Δ](conditional-expression-1)
    statement-1
_ELSEIF[Δ](conditional-expression-2)
    statement-2
_ELSE
    statement-3
_ENDIF
```

[Function]

(1) _IF ... _ENDIF
If conditional expression 1 is true, statement 1 is executed.

(2) _IF ... _ELSE ... _ENDIF
If conditional expression 1 is true, statement 1 is executed. If conditional expression 1 is false, statement 3 is executed.

(3) _IF ... _ELSEIF ... _ELSE ... _ENDIF
_ELSEIF can be coded more than once for one statement.
If conditional expression 1 is true, statement 1 is executed. If it is false, conditional expression 2 is checked. If conditional expression 2 is true, statement 2 is executed.
If conditional expression 2 is false, and if another _ELSEIF is encountered before _ENDIF, condition checking is performed for that _ELSEIF. If conditional expression 2 is false, and if _ELSEIF is not found, statement 3 is executed.

[Notes]

(1) Conditional expressions can be specified as follows:
- <FLG-type-symbol> Δrelational-operatorΔ [#]<DAT-type-symbol>
- MΔrelational-operatorΔ [#]i
- MΔrelational-operatorΔM

(a) For [#]<DAT-type-symbol> specified when <FLG-type-symbol> is included in a conditional expression, only 0 or 1 can be specified. If a value other than 0 and 1 is set for [#]<DAT-type-symbol>, an error occurs (F044: Invalid value), and expansion is not performed.

(b) Up to four conditional expressions having <FLG-type-symbol> can be coded successfully by using relational operators (OR, AND).

(c) For an explanation of other restrictions imposed on <FLG-type-symbol>, see **Section 21.2**.

(d) If a symbol defined with NIBBLEnV is specified as a general-purpose register, an error occurs. (This built-in macro does not manipulate RP.)

(e) When <FLG-type-symbol> is specified in a conditional expression, the specifiable relational operators are EQ, ==, eq, NE, !=, <>, and ne. If a relational operator other than these specifiable relational operators is specified, an error occurs.

(f) If the number of nibbles on the left side of a conditional expression is greater than the number of nibbles on the right side, the upper bits on the right side are processed as 0s.

(g) If the number of nibbles on the left side of a conditional expression is smaller than the number of nibbles on the right side, processing is performed on the assumption that the number of nibbles on the left is valid. At the same time, a warning message is output, alerting the user to the shortage of nibbles.

(h) If an illegal type is specified in the operand field, an error occurs (F045: Invalid type), and one NOP instruction is generated as an object code.

(i) If immediate data of more than eight nibbles is specified in the operand field, an error occurs (F164: The constant is over 32 bits), and one NOP instruction is generated as an object code.

(2) _IF ... _ELSE ... _ENDIF is used to control two different branches according to condition checking.

(3) _ELSEIF and _ELSE can be omitted.

[Instruction expansion]

(1) Processing for the _IF (conditional-expression-1) statement
An instruction for evaluating conditional expression 1 is generated.
An instruction for branching to _ELSEIF, _ELSE, or _ENDIF is generated.

(2) Processing for the _ELSEIF (conditional-expression-2) statement
An instruction for evaluating conditional expression 2 is generated.
An instruction for branching to the _ELSE or _ENDIF statement is generated.
A label for the branch instruction generated by the _IF statement is generated.

(3) Processing for the _ELSE statement
An instruction for branching to the _ENDIF statement is generated.
A label for the branch instruction generated by _IF or _ELSEIF is generated.

(4) Processing for the _ENDIF statement
A label for the branch instruction generated by _IF, _ELSEIF, or _ELSE is generated.

[Sample expansion]

[Example 1] When relational operator == is used in conditional expressions that compare data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
 M4     NIBBLE4    0.00H
```

```
_IF       (M4 == #0123H)
          NOP
_ELSEIF   (M4 == #3210H)
          NOP
_ELSE
          NOP
_ENDIF
```

↓

Expansion form

```
_IF  (M4 == #0123H)
```

```
SKNE   (M4)+03H,#(0123H)AND 0FH
SKE    (M4)+02H,#(0123H)SHR 4 AND 0FH
BR     ?L0
SKNE   (M4)+01H,#(0123H)SHR 8 AND 0FH
SKE    (M4),#(0123H)SHR 12 AND 0FH
?L0:
BR     ?L1
```

```
       NOP
_ELSEIF   (M4 == #3210H)
```

```
BR     ?L2
?L1:
MOV    BANK,#.DM.(M4)SHR 8 AND 0FH
SKNE   (M4)+03H,#(3210H)AND 0FH
SKE    (M4)+02H,#(3210H)SHR 4 AND 0FH
BR     ?L3
SKNE   (M4)+01H,#(3210H)SHR 8 AND 0FH
SKE    (M4),#(3210H)SHR 12 AND 0FH
?L3:
BR     ?L4
```

```
       NOP
_ELSE
```

```
BR     ?L2
?L4:
```

```
       NOP
_ENDIF
```

```
?L2:
```

[Example 2] When relational operator != is used in conditional expressions that compare data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
M4  NIBBLE4    0.00H
```

```
_IF       (M4 != #0123H)
          NOP
_ELSEIF   (M4 != #3210H)
          NOP
_ELSE
          NOP
_ENDIF
```

↓

Expansion form

```
_IF  (M4 != #0123H)
    SKE   (M4)+03H,#(0123H)AND 0FH
    BR    $+6
    SKNE  (M4)+02H,#(0123H)SHR 4 AND 0FH
    SKE   (M4)+01H,#(0123H)SHR 8 AND 0FH
    BR    $+3
    SKNE  (M4),#(0123H)SHR 12 AND 0FH
    BR    ?L0
          NOP
_ELSEIF   (M4 != #3210H)
    BR    ?L1
    ?L0:
    MOV   BANK,#.DM.(M4)SHR 8 AND 0FH
    SKE   (M4)+03H,#(3210H)AND 0FH
    BR    $+6
    SKNE  (M4)+02H,#(3210H)SHR 4 AND 0FH
    SKE   (M4)+01H,#(3210H)SHR 8 AND 0FH
    BR    $+3
    SKNE  (M4),#(3210H)SHR 12 AND 0FH
    BR    ?L2
          NOP
_ELSE
    BR    ?L1
    ?L2:
          NOP
_ENDIF
    ?L1:
```

[Example 3] When relational operator >= is used in conditional expressions that compare data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
M4     NIBBLE4    0.00H
```

```
_IF       (M4 >= #0123H)
          NOP
_ELSEIF   (M4 >= #3210H)
          NOP
_ELSE
          NOP
_ENDIF
```

↓

Expansion form

```
_IF  (M4 >= #0123H)
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    SUB    (M4)+03H,#(0123H)AND 0FH
    SUBC   (M4)+02H,#(0123H)SHR 4 AND 0FH
    SUBC   (M4)+01H,#(0123H)SHR 8 AND 0FH
    SUBC   (M4),#(0123H)SHR 12 AND 0FH
    SKF    .MF.CY SHR 4,#.DF.CY AND 0FH
    BR     ?L0
          NOP
_ELSEIF   (M4 >= #3210H)
    BR     ?L1
    ?L0:
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    MOV    BANK,#.DM.(M4)SHR 8 AND 0FH
    SUB    (M4)+03H,#(3210H)AND 0FH
    SUBC   (M4)+02H,#(3210H)SHR 4 AND 0FH
    SUBC   (M4)+01H,#(3210H)SHR 8 AND 0FH
    SUBC   (M4),#(3210H)SHR 12 AND 0FH
    SKF    .MF.CY SHR 4,#.DF.CY AND 0FH
    BR     ?L2
          NOP
_ELSE
    BR     ?L1
    ?L2:
          NOP
_ENDIF
    ?L1:
```

[Example 4] When relational operator > is used in conditional expressions that compare data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
M4     NIBBLE4    0.00H
```

```
_IF       (M4 > #0123H)
          NOP
_ELSEIF   (M4 > #3210H)
          NOP
_ELSE
          NOP
_ENDIF
```

↓

Expansion form

```
_IF  (M4 > #0123H)
```

```
OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
SUB    (M4)+03H,#(0123H)AND 0FH
SUBC   (M4)+02H,#(0123H)SHR 4 AND 0FH
SUBC   (M4)+01H,#(0123H)SHR 8 AND 0FH
SUBC   (M4),#(0123H)SHR 12 AND 0FH
SKF    .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
BR     ?L0
```

```
       NOP
_ELSEIF   (M4 > #3210H)
```

```
BR     ?L1
?L0:
OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
MOV    BANK,#.DM.(M4)SHR 8 AND 0FH
SUB    (M4)+03H,#(3210H)AND 0FH
SUBC   (M4)+02H,#(3210H)SHR 4 AND 0FH
SUBC   (M4)+01H,#(3210H)SHR 8 AND 0FH
SUBC   (M4),#(3210H)SHR 12 AND 0FH
SKF    .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
BR     ?L2
```

```
       NOP
_ELSE
```

```
BR     ?L1
?L2:
```

```
       NOP
_ENDIF
```

```
?L1:
```

[Example 5] When relational operator <= is used in conditional expressions that compare data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
M4     NIBBLE4    0.00H
```

```
_IF       (M4 <= #0123H)
          NOP
_ELSEIF   (M4 <= #3210H)
          NOP
_ELSE
          NOP
_ENDIF
```

↓

Expansion form

```
_IF  (M4 <= #0123H)
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    SUB    (M4)+03H,#(0123H)AND 0FH
    SUBC   (M4)+02H,#(0123H)SHR 4 AND 0FH
    SUBC   (M4)+01H,#(0123H)SHR 8 AND 0FH
    SUBC   (M4),#(0123H)SHR 12 AND 0FH
    SKF    .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
    SKT    (M4),#00H
    BR     ?L0
       NOP
_ELSEIF   (M4 <= #3210H)
    BR     ?L1
    ?L0:
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    MOV    BANK,#.DM.(M4)SHR 8 AND 0FH
    SUB    (M4)+03H,#(3210H)AND 0FH
    SUBC   (M4)+02H,#(3210H)SHR 4 AND 0FH
    SUBC   (M4)+01H,#(3210H)SHR 8 AND 0FH
    SUBC   (M4),#(3210H)SHR 12 AND 0FH
    SKF    .MF.CY SHR 4,#.DF.(CY OR Z)AND 0FH
    SKT    (M4),#00H
    BR     ?L2
       NOP
_ELSE
    BR     ?L1
    ?L2:
       NOP
_ENDIF
    ?L1:
```

[Example 6] When relational operator < is used in conditional expressions that compare data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
M4     NIBBLE4    0.00H
```

```
_IF       (M4 < #0123H)
          NOP
_ELSEIF   (M4 < #3210H)
          NOP
_ELSE
          NOP
_ENDIF
```

↓

Expansion form

```
_IF  (M4 < #0123H)
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    SUB    (M4)+03H,#(0123H)AND 0FH
    SUBC   (M4)+02H,#(0123H)SHR 4 AND 0FH
    SUBC   (M4)+01H,#(0123H)SHR 8 AND 0FH
    SUBC   (M4),#(0123H)SHR 12 AND 0FH
    SKT    .MF.CY SHR 4,#.DF.CY AND 0FH
    BR     ?L0
       NOP
_ELSEIF   (M4 < #3210H)
    BR     ?L1
    ?L0:
    OR     .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
    MOV    BANK,#.DM.(M4)SHR 8 AND 0FH
    SUB    (M4)+03H,#(3210H)AND 0FH
    SUBC   (M4)+02H,#(3210H)SHR 4 AND 0FH
    SUBC   (M4)+01H,#(3210H)SHR 8 AND 0FH
    SUBC   (M4),#(3210H)SHR 12 AND 0FH
    SKT    .MF.CY SHR 4,#.DF.CY AND 0FH
    BR     ?L2
       NOP
_ELSE
    BR     ?L1
    ?L2:
       NOP
_ENDIF
    ?L1:
```

[Example 7] When relational operator == is used in conditional expressions that compare data (vertical four nibbles) and immediate data (horizontal four nibbles) (with 0 set in the bank register, and 1 set in the register pointer)

```
M8V   NIBBLE8V   0.00H
R8    NIBBLE8    0.00H
M4V   NIBBLE4V   0.00H
R4    NIBBLE4    0.00H
```

```
_IF   (R8 < M8V)
      NOP
_ELSEIF (R4 < M4V)
      NOP
_ELSE
      NOP
_ENDIF
```

↓
Expansion form

```
_IF (R8 < M8V)
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    MOV    RPH,#.DM.(R8)SHR 8 AND 0FH
    AND    RPL,#01H
    OR     RPL,#.DM.(R8)SHR 3 AND 0EH
    SUB    (R8)+07H,(M8V)+070H
    SUBC   (R8)+06H,(M8V)+060H
    SUBC   (R8)+05H,(M8V)+050H
    SUBC   (R8)+04H,(M8V)+040H
    SUBC   (R8)+03H,(M8V)+030H
    SUBC   (R8)+02H,(M8V)+020H
    SUBC   (R8)+01H,(M8V)+010H
    SUBC   (R8),(M8V)
    SKT    .MF.CY SHR 4,#.DF.CY AND 0FH
    BR     ?L0
       NOP
_ELSEIF(R4 < M4V)
    BR     ?L1
    ?L0:
    OR     .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
    MOV    RPH,#.DM.(M4)SHR 8 AND 0FH
    AND    RPL,#01H
    OR     RPL,#.DM.(R4)SHR 3 AND 0EH
    MOV    BANK,#.DM.(M4V)SHR 8 AND 0FH
    SUB    (R4)+03H,#(M4V)+030H
    SUBC   (R4)+02H,#(M4V)+020H
    SUBC   (R4)+01H,#(M4V)+010H
    SUBC   (R4),(M4V)
    SKT    .MF.CY SHR 4,#.DF.CY AND 0FH
    BR     ?L2
       NOP
_ELSE
    BR     ?L1
    ?L2:
       NOP
_ENDIF
    ?L1:
```

[Example 8] When relational operator == is used in conditional expressions that compare flags and immediate data (with 0 set in the bank register)

```
F1   FLG  0.01H.0
F2   FLG  0.02H.0
F3   FLG  0.03H.0
F4   FLG  0.04H.0
```

```
_IF  ((F1 == #0)AND(F2 == #0)AND(F3 == #0)AND(F4 == #0))
     NOP
_ELSEIF   ((F1 == #1)AND(F2 == #1)AND(F3 == #1)AND(F4 == #1))
     NOP
_ELSE
     NOP
_ENDIF
```

↓

Expansion form

```
_IF  ((F1 == #0)AND(F2 == #0)AND(F3 == #0)AND(F4 == #0))
    SKF    .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
    BR     ?L0
    SKF    .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
    BR     ?L0
    SKF    .MF.(F3)SHR 4,#.DF.(F3)AND 0FH
    BR     ?L0
    SKF    .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
    BR     ?L0
      NOP
_ELSEIF   (F1 == #1)AND(F2 == #1)AND(F3 == #1)AND(F4 == #1)
    BR     ?L1
    ?L0:
    MOV    BANK,#.DF.(F1)SHR 12 AND 0FH
    SKT    .MF.(F1)SHR 4,#.DF.(F1)AND 0FH
    BR     ?L2
    SKT    .MF.(F2)SHR 4,#.DF.(F2)AND 0FH
    BR     ?L2
    SKT    .MF.(F3)SHR 4,#.DF.(F3)AND 0FH
    BR     ?L2
    SKT    .MF.(F4)SHR 4,#.DF.(F4)AND 0FH
    BR     ?L2
      NOP
_ELSE
    BR     ?L1
    ?L2:
      NOP
_ENDIF
    ?L1:
```

[Example 9] When relational operator == is used in conditional expressions that compare data (eight nibbles) and immediate data (eight nibbles) (with 0 set in the bank register)

```
M8   NIBBLE8  0.00H
M4   NIBBLE4  0.00H
```

```
_IF  (M8 == #01234567H)
     NOP
_ELSEIF   (M4 == #3210H)
     NOP
_ELSE
     NOP
_ENDIF
```

↓

Expansion form

```
_IF (M8 == #01234567H)
```

```
OR    .MF.CMP SHR 4,#.DF.(CMP OR Z)AND 0FH
SUB   (M8)+07H,#(01234567H)AND 0FH
SUBC  (M8)+06H,#(01234567H)SHR 4 AND 0FH
SUBC  (M8)+05H,#(01234567H)SHR 8 AND 0FH
SUBC  (M8)+04H,#(01234567H)SHR 12 AND 0FH
SUBC  (M8)+03H,#(01234567H)SHR 16 AND 0FH
SUBC  (M8)+02H,#(01234567H)SHR 20 AND 0FH
SUBC  (M8)+01H,#(01234567H)SHR 24 AND 0FH
SUBC  (M8),#(01234567H)SHR 28 AND 0FH
SKT   .MF.Z SHR 4,#.DF.Z AND 0FH
BR    ?L0
```

```
      NOP
_ELSEIF   (M4 == #3210H)
```

```
BR    ?L1
?L0:
MOV   BANK,#.DM.(M4)SHR 8 AND 0FH
SKNE  (M4)+03H,#(3210)AND 0FH
SKE   (M4)+02H,#(3210)SHR 4 AND 0FH
BR    ?L2
SKNE  (M4)+01H,#(3210)SHR 8 AND 0FH
SKE   (M4),#(3210)SHR 12 AND 0FH
?L2:
BR    ?L3
```

```
      NOP
_ELSE
```

```
BR    ?L1
?L3:
```

```
      NOP
_ENDIF
```

```
?L1:
```

21.4.2 _WHILE ... _ENDW

[Format]

```
_WHILE[Δ](conditional-expression)
    statement
_ENDW
```

[Function]

While the conditional expression is true, the statement is executed.

[Notes]

(1) Conditional expressions can be specified as follows:
- <FLG-type-symbol> Δrelational-operatorΔ [#]<DAT-type-symbol>
- MΔrelational-operatorΔ [#]i
- RnΔrelational-operatorΔ M

(a) For [#]<DAT-type-symbol> specified when <FLG-type-symbol> is included in a conditional expression, only 0 or 1 can be specified. If a value other than 0 or 1 is set for [#]<DAT-type-symbol>, an error occurs (F044: Invalid value), and expansion is not performed.

(b) Up to four conditional expressions having <FLG-type-symbol> can be coded successfully by using relational operators (OR, AND).

(c) For details of the other restrictions imposed on <FLG-type-symbol>, see **Section 21.2**.

(d) If a symbol defined with NIBBLEnV is specified as a general-purpose register, an error occurs. (This built-in macro does not manipulate RP.)

(e) When <FLG-type-symbol> is specified in a conditional expression, the specifiable relational operators are EQ, ==, eq, NE, !=, <>, and ne. If a relational operator other than these specifiable relational operators is specified, an error occurs.

(f) If the number of nibbles on the left side of a conditional expression is greater than the number of nibbles on the right side, the upper bits on the right side are processed as 0s.

(g) If the number of nibbles on the left side of a conditional expression is smaller than the number of nibbles on the right side, processing is performed on the assumption that the number of nibbles on the left is valid. At the same time, a warning message is output, alerting the user to the shortage of nibbles.

(h) If an illegal type is specified in the operand field, an error occurs (F045: Invalid type), and one NOP instruction is generated as an object code.

(i) If immediate data consisting of more than eight nibbles is specified in the operand field, an error occurs (F164: The constant is over 32 bits), and one NOP instruction is generated as an object code.

(2) Before the statement is executed, the conditional expression is evaluated. If the conditional expression is found to be false when evaluated for the first time, the statement is never executed.

[Instruction expansion]

(1) Processing for _WHILE (conditional-expression)
A label for the branch instruction generated by _ENDW is generated.

(2) Processing for _ENDW statement
A branch instruction for repetition is generated.
A label for the branch instruction to exit from the _WHILE block is generated.

[Sample expansion]

[Example 1] When relational operator == is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
  M4       NIBBLE4      0.00H

_WHILE     (M4 == #0123H)
       NOP
_ENDW
```

↓
Expansion form

```
_WHILE     (M4 == #0123H)
 ?L0:
 MOV       BANK,#.DM.(M4) SHR 8 AND 0FH
 SKNE      (M4)+03H, #(0123H) AND 0FH
 SKE       (M4)+02H, #(0123H) SHR 4 AND 0FH
 BR        ?L1
 SKNE      (M4)+01H, #(0123H) SHR 8 AND 0FH
 SKE       (M4), #(0123H) SHR 12 AND 0FH
 ?L1:
 BR        ?L2
          NOP
_ENDW
 BR        ?L0
 ?L2:
```

[Example 2] When relational operator != is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
  M4          NIBBLE4    0.00H

_WHILE        (M4 != #0123H)
      NOP
_ENDW
```

↓
Expansion form

```
_WHILE      (M4 != #0123H)
 ?L0:
 MOV        BANK, #.DM.(M4) SHR 8 AND 0FH
 SKE        (M4)+03H, #(0123H) AND 0FH
 BR         $+6
 SKNE       (M4)+02H, #(0123H) SHR 4 AND 0FH
 SKE        (M4)+01H, #(0123H) SHR 8 AND 0FH
 BR         $+3
 SKNE       (M4), #(0123H) SHR 12 AND 0FH
 BR         ?L1
          NOP
_ENDW
 BR         ?L0
 ?L1:
```

[Example 3] When relational operator >= is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
  M4          NIBBLE4    0.00H

_WHILE        (M4 >= #0123H)
      NOP
_ENDW
```

↓
Expansion form

```
_WHILE      (M4 >= #0123H)
 ?L0:
 OR         .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
 MOV        BANK, #.DM.(M4) SHR 8 AND 0FH
 SUB        (M4)+03H, #(0123H) AND 0FH
 SUBC       (M4)+02H, #(0123H) SHR 4 AND 0FH
 SUBC       (M4)+01H, #(0123H) SHR 8 AND 0FH
 SUBC       (M4), #(0123H) SHR 12 AND 0FH
 SKF        .MF.CY SHR 4, #.DF.CY AND 0FH
 BR         ?L1
          NOP
_ENDW
 BR         ?L0
 ?L1:
```

[Example 4] When relational operator > is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
   M4          NIBBLE4      0.00H

_WHILE        (M4 > #0123H)
        NOP
_ENDW
```

↓
Expansion form

```
    _WHILE     (M4 > #0123H)
      ?L0:
      OR        .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
      MOV       BANK, #.DM.(M4) SHR 8 AND 0FH
      SUB       (M4)+03H, #(0123H) AND 0FH
      SUBC      (M4)+02H, #(0123H) SHR 4 AND 0FH
      SUBC      (M4)+01H, #(0123H) SHR 8 AND 0FH
      SUBC      (M4), #(0123H) SHR 12 AND 0FH
      SKF       .MF.CY SHR 4, #.DF.(CY OR Z)AND 0FH
      BR        ?L1
            NOP
    _ENDW
      BR        ?L0
      ?L1:
```

[Example 5] When relational operator <= is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
   M4          NIBBLE4       0.00H

_WHILE        (M4 <= #0123H)
       NOP
_ENDW
```

↓
Expansion form

```
    _WHILE     (M4 <= #0123H)
      ?L0:
      OR        .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
      MOV       BANK, #.DM.(M4) SHR 8 AND 0FH
      SUB       (M4)+03H, #(0123H) AND 0FH
      SUBC      (M4)+02H, #(0123H) SHR 4 AND 0FH
      SUBC      (M4)+01H, #(0123H) SHR 8 AND 0FH
      SUBC      (M4), #(0123H) SHR 12 AND 0FH
      SKF       .MF.CY SHR 4, #.DF.(CY OR Z) AND 0FH
      SKT       (M4), #00H
      BR        ?L1
            NOP
    _ENDW
      BR        ?L0
      ?L1:
```

[Example 6] When relational operator < is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
    M4      NIBBLE4 0.00H
```

```
_WHILE     (M4 < #0123H)
       NOP
_ENDW
```

↓

Expansion form

```
_WHILE    (M4 < #0123H)
  ?L0:
  OR       .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
  MOV      BANK, #.DM.(M4)SHR 8 AND 0FH
  SUB      (M4)+03H, #(0123H) AND 0FH
  SUBC     (M4)+02H, #(0123H) SHR 4 AND 0FH
  SUBC     (M4)+01H, #(0123H) SHR 8 AND 0FH
  SUBC     (M4), #(0123H) SHR 12 AND 0FH
  SKT      .MF.CY SHR 4, #.DF.CY AND 0FH
  BR       ?L1
       NOP
_ENDW
  BR       ?L0
  ?L1:
```

[Example 7] When relational operator == is used in a conditional expression that compares flags and immediate data (with 0 set in the bank register)

```
    F1     FLG      0.01H.0
    F2     FLG      0.02H.0
    F3     FLG      0.03H.0
    F4     FLG      0.04H.0
```

```
_WHILE  ((F1 == #0)AND(F2 == #0)AND(F3 == #0)AND(F4 == #0))
     NOP
_ENDW
```

↓

Expansion form

```
_WHILE ((F1 == #0)AND(F2 == #0)AND(F3 == #0)AND(F4 == #0))
  ?L0:
  MOV      BANK, #.DF.(F1) SHR 12 AND 0FH
  SKF      .MF.(F1) SHR 4,#.DF.(F1) AND 0FH
  BR       ?L1
  SKF      .MF.(F2) SHR 4,#.DF.(F2) AND 0FH
  BR       ?L1
  SKF      .MF.(F3) SHR 4,#.DF.(F3) AND 0FH
  BR       ?L1
  SKF      .MF.(F4) SHR 4,#.DF.(F4) AND 0FH
  BR       ?L1
       NOP
_ENDW
  BR       ?L0
  ?L1:
```

[Example 8] When relational operator == is used in a conditional expression that compares data (eight nibbles) and immediate data (eight nibbles) (with 0 set in the bank register)

```
  M8     NIBBLE8      0.00H

_WHILE    (M8 == #01234567H)
       NOP
_ENDW
```

↓

Expansion form

```
_WHILE    (M8 == #1234567H)

  ?L0:
  OR       .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
  MOV      BANK, #.DM.(M8) SHR 8 AND 0FH
  SUB      (M8)+07H, #(01234567H) AND 0FH
  SUBC     (M8)+06H, #(01234567H) SHR 4 AND 0FH
  SUBC     (M8)+05H, #(01234567H) SHR 8 AND 0FH
  SUBC     (M8)+04H, #(01234567H) SHR 12 AND 0FH
  SUBC     (M8)+03H, #(01234567H) SHR 16 AND 0FH
  SUBC     (M8)+02H, #(01234567H) SHR 20 AND 0FH
  SUBC     (M8)+01H, #(01234567H) SHR 24 AND 0FH
  SUBC     (M8), #(01234567H) SHR 28 AND 0FH
  SKT      .MF.Z SHR 4,  #.DF.Z AND 0FH
  BR       ?L1

       NOP
_ENDW

  BR       ?L0
  ?L1:
```

21.4.3 _SWITCH ... _CASE ... _DEFAULT ... _ENDS

[Format]

```
_SWITCH[Δ](MEM-type-symbol)
_CASEΔconstant-1
    statement-1
_CASEΔconstant-2
    statement-2
      :
_CASEΔconstant-N
    statement-N
_DEFAULT
    statement-N+1
_ENDS
```

[Function]

If the value of _SWITCH matches constant i, statement i is executed (i = 1 to N). If none of constants 1 to N matches the value of α, and if _DEFAULT exists, statement N+1 is executed. If none of constants 1 to N matches the value of α, and if there is no _DEFAULT, nothing is executed.
Normally, the _BREAK statement must be specified to exit from the _SWITCH block.

[Notes]

(1) Specify a MEM-type symbol for α.

(2) After control is passed to a _CASE statement, the subsequent _CASE statements are executed sequentially. To exit from the _SWITCH block without executing the next _CASE statement, specify a _BREAK statement.

(3) Constants may be specified in binary, octal, decimal, or hexadecimal.

(4) If the same constant is specified more than once within the same _SWITCH block, an error occurs.

(5) If an instruction that generates an object code is specified between the _SWITCH statement and the first _CASE statement, an error occurs.

[Instruction expansion]

(1) Processing for _SWITCH <MEM-type-symbol>
The <MEM-type-symbol> is stored internally, then passed to _CASE. No object code is generated.

(2) Processing for _CASE constant-i
A label for the branch instruction generated by the previous _CASE statement is generated.
An instruction for comparing the symbol passed from the _SWITCH statement and constant i is generated.
When there is no _BREAK statement immediately before _CASE, a BR instruction for skipping the compare instruction is generated.

(3) Processing for the _DEFAULT statement
A label for the branch instruction generated by a _CASE statement is generated.

(4) Processing for the _ENDS statement
A label for the branch instruction generated by a _CASE or _BREAK statement is generated.

[Sample expansion]

When the _DEFAULT statement is specified (with 0 set in the bank register)

```
M1        NIBBLE4        0.00H
M5        NIBBLE4        0.38H
```

```
_SWITCH    (M1)                          ;α=M1
_CASE 1
     SUBX      M5, #1H
_DEFAULT
     SUBX      M5, #2H
_ENDS
```

↓

Expansion form

```
_SWITCH     (M1)
_CASE 1
  SKNE      (M1)+03H, #(1) AND 0FH
  SKE       (M1)+02H, #(1) SHR 4 AND 0FH
  BR        ?L0
  SKNE      (M1)+01H, #(1) SHR 8 AND 0FH
  SKE       (M1), #(1) SHR 12 AND 0FH
  ?L0:
  BR        ?L1
SUBX      M5, #1H
  MOV       BANK, #.DM.(M5) SHR 8 AND 0FH
  SUB       (M5)+03H, #(1H) AND 0FH
  SUBC      (M5)+02H, #(1H) SHR 4 AND 0FH
  SUBC      (M5)+01H, #(1H) SHR 8 AND 0FH
  SUBC      (M5), #(1H) SHR 12 AND 0FH
_DEFAULT
  ?L1:
SUBX      M5,#2H
  MOV       BANK, #.DM.(M5) SHR 8 AND 0FH
  SUB       (M5)+03H, #(2H) AND 0FH
  SUBC      (M5)+02H, #(2H) SHR 4 AND 0FH
  SUBC      (M5)+01H, #(2H) SHR 8 AND 0FH
  SUBC      (M5), #(2H) SHR 12 AND 0FH
_ENDS
```

21.4.4 _REPEAT ... _UNTIL

[Format]

```
_REPEAT
    statement
_UNTIL[Δ](conditional-expression)
```

[Format]

While the conditional expression is false, the statement is executed repeatedly.
The statement is executed at least once.

[Notes]

(1) Conditional expressions can be specified as follows:
- <FLG-type-symbol>Δrelational-operatorΔ[#]<DAT-type-symbol>
- MΔrelational-operatorΔ[#]i
- RnΔrelational-operatorΔM

(a) For [#]<DAT-type-symbol> specified when <FLG-type-symbol> is included in a conditional expression, only 0 or 1 can be specified. If a value of other than 0 or 1 is set for [#]<DAT-type-symbol>, an error occurs (F044: Invalid value), and expansion is not performed.

(b) Up to four conditional expressions having <FLG-type symbol> can be coded successfully by using relational operators (OR, AND).

(c) For details of the other restrictions imposed on <FLG-type-symbol>, see **Section 21.2**.

(d) If a symbol defined with NIBBLEnV is specified as a general-purpose register, an error occurs. (This built-in macro cannot manipulate RP.)

(e) When <FLG-type-symbol> is specified in a conditional expression, the specifiable relational operators are EQ, ==, eq, NE, !=, <>, and ne. If a relational operator other than these specifiable relational operators is specified, an error occurs.

(f) If the number of nibbles on the left side of a conditional expression is greater than the number of nibbles on the right side, the upper bits on the right side are handled as 0s.

(g) If the number of nibbles on the left side of a conditional expression is smaller than the number of nibbles on the right side, processing is performed assuming the number of nibbles on the left to be valid. At the same time, a warning message is output, alerting the user to the shortage of nibbles.

(h) If an illegal type is specified in the operand field, an error occurs (F045: Invalid type), and one NOP instruction is generated as an object code.

(i) If immediate data of more than eight nibbles is specified in the operand field, an error occurs (F164: The constant is over 32 bits), and one NOP instruction is generated as an object code.

(2) After the statement is executed once, the conditional expression is evaluated.

[Instruction expansion]

(1) Processing for the _REPEAT statement
A label for the branch instruction generated by the _UNTIL statement is generated.

(2) Processing for the _UNTIL (conditional-expression) statement
An instruction for evaluating the conditional expression is generated.

[Sample expansion]

[Example 1] When relational operator == is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
M4      NIBBLE4      0.00H
```

```
_REPEAT
       NOP
_UNTIL      (M4 == #0123H)
```

↓

Expansion form

```
_REPEAT
  ?L0:

            NOP
_UNTIL       (M4 == #0123H)

  MOV       BANK, #.DM.(M4) SHR 8 AND 0FH
  SKNE      (M4)+03H, #(0123H) AND 0FH
  SKE       (M4)+02H, #(0123H) SHR 4 AND 0FH
  BR        ?L1
  SKNE      (M4)+01H, #(0123H) SHR 8 AND 0FH
  SKE       (M4), #(0123H) SHR 12 AND 0FH
  ?L1:
  BR        ?L0
```

[Example 2] When relational operator != is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
   M4      NIBBLE4      0.00H

_REPEAT
       NOP
_UNTIL      (M4 != #0123H)
```

↓
Expansion form

```
_REPEAT
 ?L0:

            NOP

_UNTIL       (M4 != #0123H)
 MOV       BANK, #.DM.(M4) SHR 8 AND 0FH
 SKE       (M4)+03H, #(0123H) AND 0FH
 BR        $+6
 SKNE      (M4)+02H, #(0123H) SHR 4 AND 0FH
 SKE       (M4)+01H, #(0123H) SHR 8 AND 0FH
 BR        $+3
 SKNE      (M4), #(0123H) SHR 12 AND 0FH
 BR        ?L0
```

[Example 3] When relational operator >= is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
   M4      NIBBLE4      0.00H

_REPEAT
       NOP
_UNTIL       (M4 >= #0123H)
```

↓
Expansion form

```
_REPEAT
 ?L0:

            NOP
_UNTIL        (M4 >= #0123H)
 OR        .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
 MOV       BANK, #.DM.(M4)SHR 8 AND 0FH
 SUB       (M4)+03H, #(0123H) AND 0FH
 SUBC      (M4)+02H, #(0123H) SHR 4 AND 0FH
 SUBC      (M4)+01H, #(0123H) SHR 8 AND 0FH
 SUBC      (M4), #(0123H) SHR 12 AND 0FH
 SKF       .MF.CY SHR 4, #.DF.CY AND 0FH
 BR        ?L0
```

[Example 4] When relational operator > is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
    M4      NIBBLE4       0.00H

_REPEAT
        NOP
_UNTIL      (M4 > #0123H)
```

↓
Expansion form

```
_REPEAT
  ?L0:
            NOP
_UNTIL        (M4 > #0123H)
  OR        .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
  MOV       BANK, #.DM.(M4) SHR 8 AND 0FH
  SUB       (M4)+03H, #(0123H) AND 0FH
  SUBC      (M4)+02H, #(0123H) SHR 4 AND 0FH
  SUBC      (M4)+01H, #(0123H) SHR 8 AND 0FH
  SUBC      (M4), #(0123H) SHR 12 AND 0FH
  SKF       .MF.CY SHR 4, #.DF.(CY OR Z) AND 0FH
  BR        ?L0
```

[Example 5] When relational operator <= is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
    M4      NIBBLE4       0.00H

_REPEAT
        NOP
_UNTIL      (M4 <= #0123H)
```

↓
Expansion form

```
_REPEAT
  ?L0:
            NOP
_UNTIL        (M4 <= #0123H)
  OR        .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
  MOV       BANK, #.DM.(M4) SHR 8 AND 0FH
  SUB       (M4)+03H, #(0123H) AND 0FH
  SUBC      (M4)+02H, #(0123H) SHR 4 AND 0FH
  SUBC      (M4)+01H, #(0123H) SHR 8 AND 0FH
  SUBC      (M4), #(0123H) SHR 12 AND 0FH
  SKF       .MF.CY SHR 4, #.DF.(CY OR Z) AND 0FH
  SKT       (M4). #00H
  BR        ?L0
```

[Example 6] When relational operator < is used in a conditional expression that compares data (four nibbles) and immediate data (four nibbles) (with 0 set in the bank register)

```
   M4     NIBBLE4      0.00H
```

```
_REPEAT
       NOP
_UNTIL      (M4 < #0123H)
```

↓

Expansion form

```
_REPEAT
 ?L0:
            NOP
_UNTIL       (M4 < #0123H)
 OR         .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
 MOV        BANK, #.DM.(M4) SHR 8 AND 0FH
 SUB        (M4)+03H, #(0123H) AND 0FH
 SUBC       (M4)+02H, #(0123H) SHR 4 AND 0FH
 SUBC       (M4)+01H, #(0123H) SHR 8 AND 0FH
 SUBC       (M4), #(0123H) SHR 12 AND 0FH
 SKT        .MF.CY SHR 4, #.DF.CY AND 0FH
 BR         ?L0
```

[Example 7] When relational operator == is used in a conditional expression that compares flags and immediate data (with 0 set in the bank register)

```
  F1     FLG          0.01H.0
  F2     FLG          0.02H.0
  F3     FLG          0.03H.0
  F4     FLG          0.04H.0
```

```
_REPEAT
       NOP
_UNTIL     ((F1 == 0)AND(F2 == 0)AND(F3 == 0)AND(F4 == 0))
```

↓

Expansion form

```
_REPEAT
 ?L0:
          NOP
_UNTIL     ((F1 == 0)AND(F2 == 0)AND(F3 == 0)AND(F4 == 0))
 MOV        BANK, #.DF.(F1) SHR 12 AND 0FH
 SKF        .MF.(F1) SHR 4, #.DF.(F1) AND 0FH
 BR         ?L1
 SKF        .MF.(F2) SHR 4, #.DF.(F2) AND 0FH
 BR         ?L1
 SKF        .MF.(F3) SHR 4, #.DF.(F3) AND 0FH
 BR         ?L1
 SKF        .MF.(F4) SHR 4, #.DF.(F4) AND 0FH
 ?L1:
 BR         ?L0
```

[Example 8] When relational operator == is used in a conditional expression that compares data (eight nibbles) and immediate data (eight nibbles) (with 0 set in the bank register)

```
   M8      NIBBLE8     0.00H
```

```
_REPEAT
       NOP
_UNTIL      (M8 == #01234567H)
```

↓

Expansion form

```
_REPEAT
  ?L0:
         NOP
_UNTIL    (M8 == #01234567H)
  OR        .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
  MOV       BANK, #. DM.(M8)SHR 8 AND 0FH
  SUB       (M8)+07H, #(01234567H) AND 0FH
  SUBC      (M8)+06H, #(01234567H) SHR 4 AND 0FH
  SUBC      (M8)+05H, #(01234567H) SHR 8 AND 0FH
  SUBC      (M8)+04H, #(01234567H) SHR 12 AND 0FH
  SUBC      (M8)+03H, #(01234567H) SHR 16 AND 0FH
  SUBC      (M8)+02H, #(01234567H) SHR 20 AND 0FH
  SUBC      (M8)+01H, #(01234567H) SHR 24 AND 0FH
  SUBC      (M8), #(01234567H) SHR 28 AND 0FH
  SKT       .MF.Z SHR 4, #.DF. Z AND 0FH
  BR        ?L0
```

21.4.5 _FOR_ ... _NEXT

[Format]

```
_FOR[Δ](expression-1:(expression-2):expression-3)
     statement
_NEXT
```

[Function]

An initial value is set in expression 1. While a conditional expression specified for expression 2 is true, the statement and expression 3 are executed.

[Notes]

Specify an initial value (assignment expression) for expression 1. For expression 2, specify a conditional expression, and for expression 3, specify an assignment expression such as an increment or decrement expression.

[Instruction expansion]

(1) Processing for the _FOR (expression-1:(expression-2):expression-3) statement
An assignment instruction for the assignment expression in expression 1, an instruction for evaluating the conditional expression specified in expression 2, and a label for the branch instruction generated by the _NEXT statement are generated. A branch instruction for exiting from the _FOR block is also generated.

(2) Processing for the _NEXT statement
An assignment instruction for the assignment expression in expression 3, an instruction for branching to the _FOR statement, and a label for exiting from the _FOR block are generated.

[Sample expansion]

[Example 1] When relational operator == is used, and data (eight nibbles) and immediate data (eight nibbles) are compared (with 0 set in the bank register)

```
M1    NIBBLE4       0.00H
M4    NIBBLE4       0.38H
```

```
_FOR (M1 = #0123H: (M1 != #3210H): M1 ++)
       ADDCX M4, #1H
_NEXT
```

↓

Expansion form

```
_FOR (M1 = #0123H: (M1 != #3210H): M1 ++)
  MOV      (M1)+03H, #(0123H) AND 0FH
  MOV      (M1)+02H, #(0123H) SHR 4 AND 0FH
  MOV      (M1)+01H, #(0123H) SHR 8 AND 0FH
  MOV      (M1), #(0123H) SHR 12 AND 0FH
  ?L0:
  MOV      BANK, #.DM.(M1) SHR 8 AND 0FH
  SKE      (M1)+03H, #(3210H) AND 0FH
  BR       $+6
  SKNE     (M1)+02H, #(3210H) SHR 4 AND 0FH
  SKE      (M1)+01H, #(3210H) SHR 8 AND 0FH
  BR       $+3
  SKNE     (M1), #(3210H) SHR 12 AND 0FH
  BR       ?L1
     ADDCX  M4, #1H
  ADDC     (M4)+03H, #(1H) AND 0FH
  ADDC     (M4)+02H, #(1H) SHR 4  AND 0FH
  ADDC     (M4)+01H, #(1H) SHR 8  AND 0FH
  ADDC     (M4), #(1H) SHR 12  AND 0FH
_NEXT
  ADDC     (M1)+03H, #(1) AND 0FH
  ADDC     (M1)+02H, #(1) SHR 4  AND 0FH
  ADDC     (M1)+01H, #(1) SHR 8  AND 0FH
  ADDC     (M1), #(1) SHR 12  AND 0FH
  BR       ?L0
  ?L1:
```

21.4.6 _BREAK

[Format]

_BREAK

[Function]

_BREAK terminates the execution of the innermost _WHILE, _REPEAT, _FOR, or _SWITCH block where _BREAK is located.

[Instruction expansion]

A branch instruction for exiting from the _WHILE, _REPEAT, _FOR, or _SWITCH block is generated.

[Sample expansion]

[Example 1] When exiting from the _SWITCH block (with 0 set in the bank register)

```
M1        NIBBLE4      0.00H
M5        NIBBLE4      0.38H
```

```
_SWITCH     (M1)     ;α=M1
     _CASE 1
        SUBX     M5 ,#1H
        _BREAK        ;Exiting from the _SWITCH block here.
     _DEFAULT
        SUBX     M5 ,#2H
_ENDS
```

↓

Expansion form

```
_SWITCH      (M1)
_CASE 1
  SKNE      (M1)+03H, #(1) AND 0FH
  SKE       (M1)+02H, #(1) SHR 4 AND 0FH
  BR        ?L0
  SKNE      (M1)+01H, #(1) SHR 8 AND 0FH
  SKE       (M1), #(1)SHR 12 AND 0FH
  ?L0:
  BR        ?L1
SUBX      M5 ,#1H
  MOV       BANK, #.DM.(M5) SHR 8 AND 0FH
  SUB       (M5)+03H, #(1H) AND 0FH
  SUBC      (M5)+02H, #(1H) SHR 4 AND 0FH
  SUBC      (M5)+01H, #(1H) SHR 8 AND 0FH
  SUBC      (M5), #(1H) SHR 12 AND 0FH
_BREAK
  BR        ?L2
_DEFAULT
  ?L1:
SUBX      M5 ,#2H
  MOV       BANK, #.DM.(M5) SHR 8 AND 0FH
  SUB       (M5)+03H, #(2H) AND 0FH
  SUBC      (M5)+02H, #(2H) SHR 4 AND 0FH
  SUBC      (M5)+01H, #(2H) SHR 8 AND 0FH
  SUBC      (M5), #(2H)SHR 12 AND 0FH
_ENDS
  ?L2:
```

[Example 2] When exiting from the _WHILE block (with 0 set in the bank register)

```
M1      NIBBLE4    0.00H
M5      NIBBLE4    0.38H

_WHILE    (M1 NE M5)
    _IF     (M1 != #0FH)
         ADDCX    M1, #1H
    _ELSE
          _BREAK          ;Exiting from the _WHILE block here.
    _ENDIF
_ENDW
```

↓

Expansion form

```
_WHILE      (M1 NE M5)
 ?L0:
 OR         .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
 MOV        RPH, #.DM.(M1)SHR 8 AND 0FH
 AND        RPL, #01H
 OR         RPL, #.DM.(M1) SHR 3 AND 0EH
 MOV        BANK, #.DM.(M5) SHR 8 AND 0FH
 SUB        (M1)+03H, #(M5)+03H
 SUBC       (M1)+02H, #(M5)+02H
 SUBC       (M1)+01H, #(M5)+01H
 SUBC       (M1), (M5)
 SKF        .MF.Z SHR 4, #.DF.Z AND 0FH
 BR         ?L1
_IF      (M1 != #0FH)
 SKE        (M1)+03H, #(0FH) AND 0FH
 BR         $+6
 SKNE       (M1)+02H, #(0FH) SHR 4 AND 0FH
 SKE        (M1)+01H, #(0FH) SHR 8 AND 0FH
 BR         $+3
 SKNE       (M1), #(0FH) SHR 12 AND 0FH
 BR         ?L2
ADDCX     M1, #1H
 ADDC       (M1)+03H, #(1H) AND 0FH
 ADDC       (M1)+02H, #(1H) SHR 4 AND 0FH
 ADDC       (M1)+01H, #(1H) SHR 8 AND 0FH
 ADDC       (M1), #(1H) SHR 12 AND 0FH
_ELSE
 BR         ?L3
 ?L2:
_BREAK
 BR         ?L1
_ENDIF
 ?L3:
_ENDW
 BR         ?L0
 ?L1:
```

21.4.7 _CONTINUE

[Format]

```
_CONTINUE
```

[Function]

_CONTINUE causes an unconditional branch to the top label of the innermost _WHILE, _REPEAT, or _FOR block where _CONTINUE is located, and executes the next loop.

[Instruction expansion]

A branch instruction for repeating a loop of the _WHILE, _REPEAT, or _FOR block is generated.

```
BR  xxxx  (xxxx:  Branch destination label)
```

[Sample expansion]

When _CONTINUE is specified in a _WHILE block (with 0 set in the bank register)

```
M1      NIBBLE4     0.00H
M5      NIBBLE4     0.38H

_WHILE    (M1 NE M5)
    SUBX    M1, #1H
    _IF     (M1 >= #0FFH)
            _CONTINUE          ;Branch to the top label of _WHILE
    _ELSE
        SUBX     M1, #1H
    _ENDIF
_ENDW
```

↓

Expansion form

```
_WHILE    (M1 NE M5)
 ?L0:
 OR       .MF.CMP SHR 4, #.DF.(CMP OR Z) AND 0FH
 MOV      RPH, #.DM.(M1) SHR 8 AND 0FH
 AND      RPL, #01H
 OR       RPL, #.DM.(M1) SHR 3 AND 0EH
 MOV      BANK, #.DM.(M5) SHR 8 AND 0FH
 SUB      (M1)+03H,#(M5)+03H
 SUBC     (M1)+02H,#(M5)+02H
 SUBC     (M1)+01H,#(M5)+01H
 SUBC     (M1), (M5)
 SKF      .MF.Z SHR 4, #.DF.Z AND 0FH
 BR       ?L1
SUBX    M1, #1H
 SUB      (M1)+03H, #(1H) AND 0FH
 SUBC     (M1)+02H, #(1H) SHR 4 AND 0FH
 SUBC     (M1)+01H, #(1H) SHR 8 AND 0FH
 SUBC     (M1), #(1H) SHR 12 AND 0FH
_IF     (M1 >= #0FFH)
 OR       .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
 SUB      (M1)+03H, #(0FFH) AND 0FH
 SUBC     (M1)+02H, #(0FFH) SHR 4 AND 0FH
 SUBC     (M1)+01H, #(0FFH) SHR 8 AND 0FH
 SUBC     (M1), #(0FFH) SHR 12 AND 0FH
 SKF      .MF.CY SHR 4, #.DF.CY AND 0FH
 BR       ?L2
_CONTINUE
 BR       ?L0
_ELSE
 BR       ?L3
 ?L2:
SUBX    M1, #1H
 MOV      BANK, #.DM.(M1) SHR 8 AND 0FH
 SUB      (M1)+03H, #(1H) AND 0FH
 SUBC     (M1)+02H, #(1H) SHR 4 AND 0FH
 SUBC     (M1)+01H, #(1H) SHR 8 AND 0FH
 SUBC     (M1), #(1H) SHR 12 AND 0FH
_ENDIF
 ?L3:
_ENDW
 BR       ?L0
 ?L1:
```

21.4.8 _GOTO

[Format]

```
_GOTOΔ<branch-destination-label>
```

[Function]

_GOTO branches to the branch destination label unconditionally.

[Notes]

(1) Specify the _GOTO statement when error handling must be performed immediately by a program such as an error handling program, or when errors may occur in different locations, and they are handled in the same manner.

(2) For the label, specify a symbol specified in the label field in assembly language.

[Instruction expansion]

An instruction to branch to the <branch-destination-label> is generated.

```
BRXΔ<branch-destination-label>
```

[Sample expansion]

When exiting from the _WHILE block (with 0 set in the bank register)

```
M1          NIBBLE4          0.00H
M5          NIBBLE4          0.38H

_WHILE     (M1 NE M5)
      _IF    (M1 != #0FH)
         ADDCX    M1, #1H
      _ELSE
            _GOTO  L1             ;Branch to L1 unconditionally
      _ENDIF
_ENDW
L1:
```

↓

Expansion form

```
_WHILE      (M1 NE M5)
  ?L0:
  OR         .MF.CMP SHR 4, #.DF.(CMP OR Z)AND 0FH
  MOV        RPH, #.DM.(M1) SHR 8 AND 0FH
  AND        RPL, #01H
  OR         RPL, #.DM.(M1) SHR 3 AND 0EH
  MOV        BANK,#.DM.(M5) SHR 8 AND 0FH
  SUB        (M1)+03H, (M5)+03H
  SUBC       (M1)+02H, (M5)+02H
  SUBC       (M1)+01H, (M5)+01H
  SUBC       (M1), (M5)
  SKF        .MF.Z SHR 4, #.DF.Z AND 0FH
  BR         ?L1
_IF      (M1 != #0FH)
  SKE        (M1)+03H, #(0FH) AND 0FH
  BR         $+6
  SKNE       (M1)+02H, #(0FH) SHR 4 AND 0FH
  SKE        (M1)+01H, #(0FH) SHR 8 AND 0FH
  BR         $+3
  SKNE       (M1) ,#(0FH) SHR 12 AND 0FH
  BR         ?L2
ADDCX      M1, #1H
  ADDC       (M1)+03H, #(1H) AND 0FH
  ADDC       (M1)+02H, #(1H) SHR 4 AND 0FH
  ADDC       (M1)+01H, #(1H) SHR 8 AND 0FH
  ADDC       (M1), #(1H) SHR 12 AND 0FH
_ELSE
  BR         ?L3
  ?L2:
_GOTO      L1
  BR         (L1)
_ENDIF
  ?L3:
_ENDW
  BR         ?L0
  ?L1:
L1:
```

[MEMO]

CHAPTER 22 OPERATING PROCEDURES

22.1 FILE CONFIGURATION

The RA17K assembler package consists of the following files:

RA17K assembler package file configuration

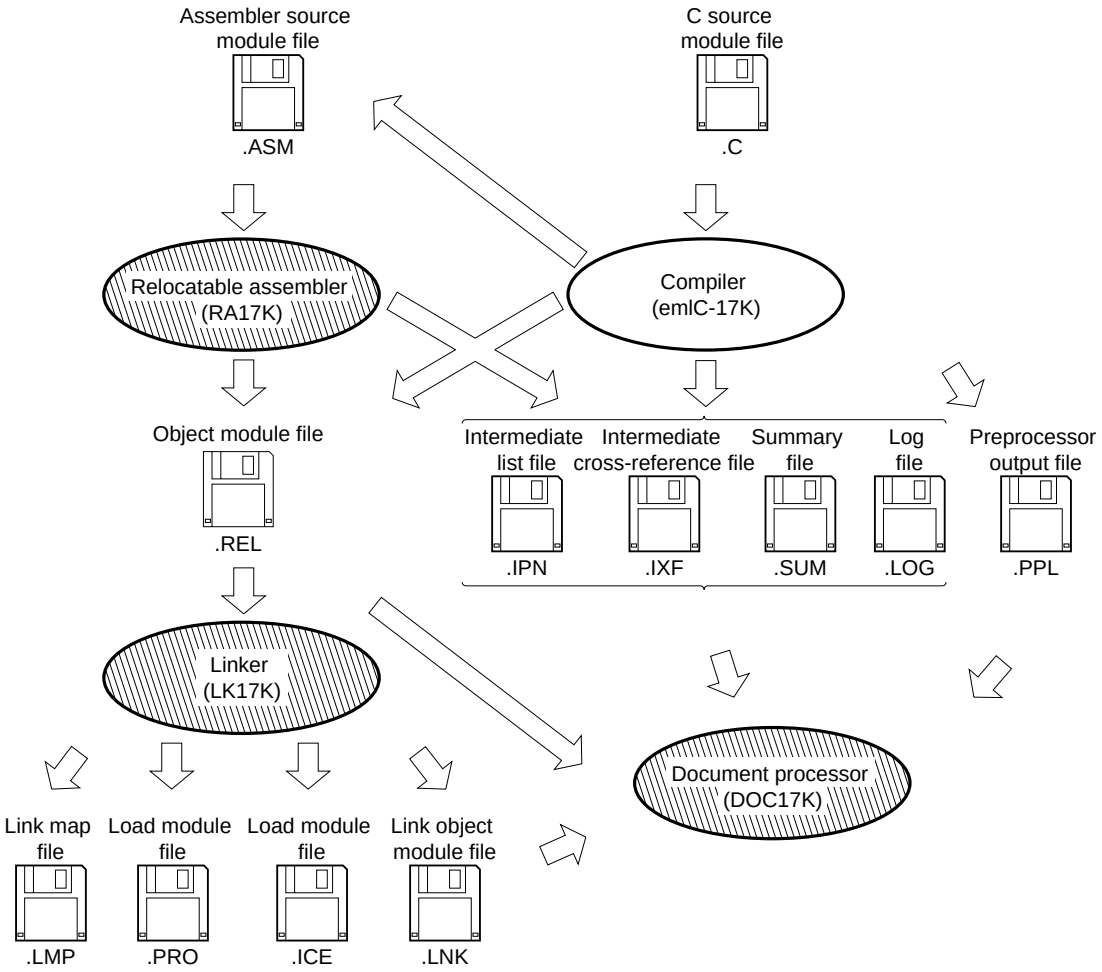
- **Assembler**
  RA17K.EXE : Main body of the assembler
- **Linker**
  LK17K.EXE : Main body of the linker
- **Make processor**
  MAKE.EXE : Main body of the make processor
  CNV17K.EXE : Sequence file converter
- **Document processor**
  DOC17K.EXE : Main body of the document processor
- **Document processor Utilities**
  LIST17K.EXE : Absolute-address-based list output utility
  XREF17K.EXE : Cross-reference list output utility
  PUB17K.EXE : Public cross-reference output utility
  MAP17K.EXE : Module map list output utility
  AMAP17K.EXE : Program map list output utility
  REP17K.EXE : Report list output utility
  SPEC17K.EXE : Document list output utility
  MOD17K.EXE : Module list output utility
  TREE17K.EXE : Call tree list output utility

Caution In addition to the files listed above, README.DOC is sometimes provided. When provided, it should be read before the assembler is used, because it contains information not provided in the manual.

Remark The document processor utilities are started by the document processor (DOC17K.EXE). The user cannot start them separately.

The outlined flow of 17K series software development is shown below.

Figure 22-1. Outlined Flow of 17K Series Software Development



Remarks 1. Refer to the **Document Processor (DOC17K) User's Manual** for files created by the document processor .

2. The programs indicated by hatching are included in the relocatable assembler package.

22.2 INSTALLATION

RA17K must be installed on a hard disk before it can be used. For details, see **Section 1.8**.

22.3 STARTUP

RA17K is started using the following command:

```
A>[path-name]RA17KΔdevice-file-nameΔsource-module-file-nameΔ[option-list]
```

The underlined entities are what the user enters. The command name (RA17K), device file name, source module file name, and each option described in the option list must be separated with at least one space character.

- **device-file-name**
  This parameter specifies the file that holds definitions of information specific to the target device. The device file name varies with the target device. If a file name extension is omitted, .DEV is assumed. If there is an .OPT file, it must be placed in the same directory as the .DEV file.

- **source-module-file-name**
  This parameter specifies the file that holds an assembly source program. Any file-name extensions except .DEV, .SEQ, and .OPT can be used. The file name extension need not necessarily be .ASM. If a file name extension is omitted from the source module file, .ASM is assumed. This holds true of include files, which are read into a source program.

- **option-list**
  See **Section 22.6**.

22.3.1 Entering a Device File Name

[Explanation]

(1) The device file name must follow the character string RA17K. In addition, there must be at least one space character between the character string and the device file name.

(2) The extension of the device file name must be .DEV, unless omitted. If omitted, .DEV is assumed.

(3) The device file name can contain a path name where the device file is located. It can consist of up to 141 characters.

Example: `A: \UPD17000\D17001.DEV`

[Notes]

(1) If the specified device file is not found, an error (A104: Can't open file <filename>) is detected, and assembly is aborted (where <filename> is the specified device file name).

(2) If a file name extension other than .DEV is specified, an error (A105: <filename> is not device file) is detected, and assembly is aborted (where <filename> is the specified device file name).

22.3.2 Entering a Source Module File Name

[Explanation]

(1) The source module file name must follow the device file name. In addition, there must be at least one space character between the device file name and the source module file name.

(2) The extension of the source module file can be omitted. If omitted, .ASM is assumed.

(3) The source module file name can contain a path name where the source module file is located. It can consist of up to 141 characters.

Example: `A: \UPD17000\D17001.ASM`

[Notes]

(1) If the specified source module file is not found, an error (A104: Can't open file <filename>) is detected, and assembly is aborted (where <filename> is the specified source module file name).

(2) If no source module file name is specified, an error (A158: Parameter error at start time) is detected.

(3) There must be at least one space character between the source module file name and the option specification that follows it.

(4) If the file name extension of the specified file is .DEV, .OPT, or .SEQ, an error (A163: <filename> is not a source file) is detected.

22.3.3 Entering Options

[Explanation]

(1) The option list must follow the source module file name. In addition, there must be at least one space character between the source module file name and the option list.

(2) The description of an option begin with a hyphen (-).

(3) If the option list contains more than one option description, they must be separated with a hyphen. Do not place a space character between each option description and the corresponding hyphen, or an error (A106: Invalid option (option_name)) is detected.

(4) If an attempt is made to specify mutually exclusive options, the last option to be specified is accepted.

[Notes]
If an attempt is made to specify an invalid option, an error (A106: Invalid option (option_name)) is detected.

Example 1: `RA17K D17001.DEV TEST.ASM -OBJ -LIST` This is a valid specification.
Example 2: `RA17K D17001.DEV TEST.ASM -OBJ-LIST`

An error is detected because there is no space between -OBJ and -LIST.

Example 3: `RA17K D17001.DEV TEST.ASM -OBJ - LIST`

An error is detected because there is space between the hyphen and LIST.

22.3.4 If All Parameters Are Omitted; Only RA17K Is Specified

If all parameters are omitted, the following help message is displayed, and the DOS prompt appears again.

```
usage : RA17K device-file input-file[option[···]]
The file is as follows.
device-file          :Specify 17K series device file name.
                      if extension omitted, .DEV assumed.
input-file           :Specify assembler source module fine name.
                      if extension omitted, .ASM assumed.

The option is as follows([]means omissible).
–ABS                 :Set absolute assembler mode.
–HOS/–NOH            :Use SIMPLEHOST / Not.
–SUM/–NOS            :Create summary file / Not.
–INC[=path–name]     :Specify path name list.
-LIS[=path–name]/-NOL        :Create assemble list file / Not.
–OBJ[=directory–name]/–NOO   :Create rel file / Not.
–XRE[=directory–name]/–NOX   :Create cross reference file / Not.
–UND[=directory–name]/–NOU   :Create underfined symbol file / Not.
–WAR=n               :Set warning level.
–WOR=path_name       :Set temporary directory.
–ZZZn=m              :Set assembler parameter.
–TAGS="string"       :Set tag start code.
–TAGE="string"       :Set tag end code.

DEFAULT ASSIGNMENT: –NOH –NOS –LIS –OBJ –XRE –NOU –WAR=0
```

22.4 STARTUP AND END MESSAGES

(1) Startup message

The startup message echoes back exactly the source module file name entered by the user.

```
17K Series Relocatable Assembler V1.xx [DD MMM YY]
    Copyright(C)NEC Corporation 19xx

--- Assemble start hh:mm:ss yy/dd/mm ---
Device file name: Name of the device file
Source file name: Name of the source module file
```

Vx.xx	: Version No.
[DD MMM YY]	: Date of the current release
19xx	: Year of the first release
hh:mm:ss yy/dd/mm	: Time and date on which the assembly started

(2) End message

The end message contains the time and date on which the assembly ended. If errors and/or warnings are detected during assembly, their totals are displayed at the end of assembly.

Caution After 65535 errors or warnings are detected, the count returns to 0. If 65536 errors or warnings are detected, for example, the total count is indicated as 0. Moreover, the end message contains the status of memory used, and the names and sizes of created files (if any).

- If neither an error nor a warning is detected during assembly, the following end message appears:

```
--- Assemble end hh:mm:ss yy/dd/mm ---
Total error(s):0 Total warning(s):0
```

- If errors and warnings are detected during assembly, their counts are displayed.

```
--- Assemble end hh:mm:ss yy/dd/mm ---
Total error(s):5 Total warning(s):2
```

- If a fatal error (or abort error) occurs during assembly and disables further processing from continuing, the following message is displayed, and assembly is aborted.

```
--- Assemble end hh:mm:ss yy/dd/mm ---
program aborted
```

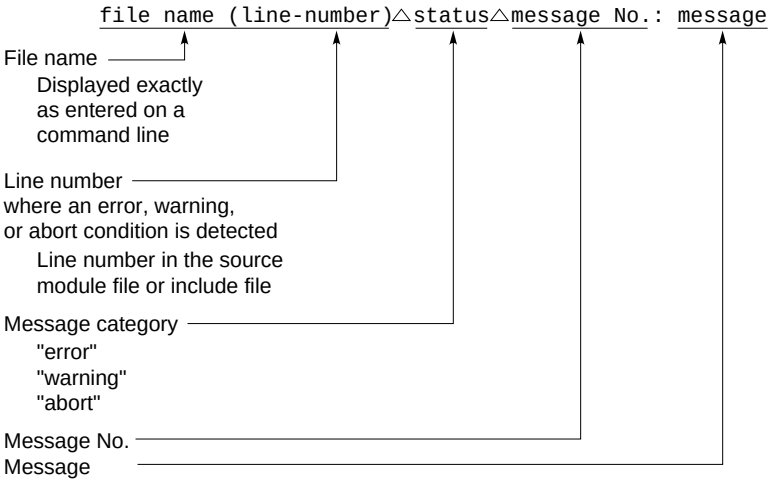
22.5 MESSAGES DISPLAYED DURING ASSEMBLY

(1) Error, warning, and abort messages

The messages displayed during assembly include error, warning, and abort messages. See **Chapter 24** for each message and their meanings.

[Output format]

If an error, warning, or abort condition occurs during assembly, the relevant message, as well as the source line containing the error, warning, or abort condition, is displayed in the format described below.



[Explanation]

- Error message
  An error message is displayed, mainly if an error is detected in a description in the source module file, but assembly can continue. When an error message is displayed, only the relevant source line is assumed to be invalid. If the source line is one that generates an object code, a NOP instruction is generated, and assembly is continued. If an error message is output, an object module file is not created. The following table lists whether each output file is created.

Output file name	Whether created or not
Object module file (.REL)	Not created
Log file (.LOG)	Created
Symbol file (.SYM)	
Summary file (.SUM)	
Undefined symbol file (.UND)	
Intermediate list file (.IPN)	
Intermediate cross-reference file (.IXF)	

- Warning message
  If there is a possibility that the program does not behave as intended, a message is displayed to warn. Even if a warning message is displayed, assembly continues. When a warning message is displayed, all output files are created normally, provided that no error has occurred.

- Abort message
  If assembly cannot continue because of an error (fatal error), an abort message is displayed. When an abort message is displayed, assembly ends immediately. If a fatal error occurs, all output files except the log and intermediate list files are not created normally. The log and intermediate list files hold information collected before the fatal error occurs.

(2) Messages indicating the status of memory used and created files

Messages are displayed to indicate the size of memory reserved, and files created, by RA17K.

- Format of the memory use status report message

```
MEM AREA = XXXXXXXXXXXXXX : YYYYYY : ZZZZZZ
```

xxxxxxxxxxxxxx is the name of a memory area.
The following types of memory areas are supported. (No special order is observed.)

REL	TMP	Temporary file for a REL file
SUMMARY	TMP	Temporary file for a summary file
OPTION	TMP	Temporary file for the OPTION pseudo instruction
INTERM	TMP	Temporary file for intermediate code
LIST	FILE	Intermediate list file
XREF	FILE	Cross-reference file
REL	FILE	REL file
SUMMARY	FILE	Summary file
SYMBOL	MAIN	Symbol table
SYMBOL	EMS	EMS
ETC		Others

YYYYYY represents the size of a reserved memory area (in bytes in decimal notation).
ZZZZZZ represents the size of an actually used memory area (in bytes in decimal notation).

- Format of the file creation report message

```
FILE NAME = AAAAAAAA.BBB: CCCCCC
```

AAAAAAAA.BBB is the name of a created file (full path name).
CCCCCC represents the size of a created file (in bytes in decimal notation).

[Output example]

```
17K Series Relocatable Assembler V1.xx [DD MMM YY]
    Copyright(C)NEC Corporation 19xx

--- Assemble start hh:mm:ss yy/dd/mm ---
Device file name: a.DEV
Source file name: a.ASM

MEM AREA   = REL       TMP   :  4317 :    259
MEM AREA   = REL       TMP   :  4317 :    229
MEM AREA   = REL       TMP   :  4317 :   4317
FILE NAME  = E:\r$b05938 :   12000
MEM AREA   = REL       TMP   :  4317 :   4317
FILE NAME  = E:\r$a05938 :   78000
MEM AREA   = REL       TMP   :  4317 :    225
MEM AREA   = REL       TMP   :  2269 :   2269
FILE NAME  = E:\r$c05938 :  120440
MEM AREA   = REL       TMP   :  4317 :    260
MEM AREA   = REL       FILE  :  4321 :   4321
FILE NAME  = ********.REL :  180656
MEM AREA   = INTERM    TMP   :  8413 :   8413
FILE NAME  = E:\r$005938 :  159021
MEM AREA   = XREF      FILE  :  8417 :   8417
FILE NAME  = ********.IXF :   48000
MEM AREA   = SYMBOL    MAIN  :  46592:  46592
MEM AREA   = SYMBOL    EMS   : 114688: 114688
MEM AREA   = LIST      TMP   :   4321:   4321
FILE NAME  = ********.IPN :  443306
MEM AREA   = ETC             :   5223:    404

--- Assemble end hh:mm:ss yy/dd/mm ---

Total error(s):0 Total warning(s):0
```

22.6 ASSEMBLER OPTIONS

[Notes]

(1) For options for specifying numbers, only binary, decimal, and hexadecimal numbers can be specified. A description of a hexadecimal number must begin with a numeral from 0 to 9. If an attempt is made to specify any other numeral as the first numeral of a hexadecimal number, an error (A106: Invalid option (option_name)) is detected.

(2) An option can be described in either uppercase or lowercase characters.

(3) If the description of an option contains only a path-name, the directory name need not end with the special character "\".

Example: Outputting an object file to the \RA17K path

```
-OBJ = \RA17K\
-OBJ = \RA17K
```

Both descriptions are acceptable.

When the format of <path-name>\<file-name> is used, it is assumed to be a description of <path-name> alone, if the last character is "\".

Options

Option name	Description	Default	Assumption used when -HOST is specified	Reference page
-OBJ[ECT][=<path-name>] -NOO[BJECT]	Controls object module file (.REL) output.	-OBJ	-OBJ is specified forcibly; the user's specification is ignored.	p.443
-LIS[T][=<file-name>] -NOL[IST]	Controls cross-reference list file (.IPN) output.	-LIS	-LIS is specified forcibly; the user's specification is ignored.	p.444
-UND[EF][=<file-name>] -NOU[NDEF]	Controls undefined symbol file (.UND) output.	-NOU	As specified	p.445
-WOR[K][=<path-name>]	Specifies the path name of a work drive.	No specification	As specified	p.446
-HOS[T] -NOH[OST]	Controls information about *SIMPLEHOST*.	-NOH	-HOST	p.447
-ZZZn = m (0 ≤ n ≤ 15) (0 ≤ m ≤ FFFFFFFFH)	Sets the initial values of assemble time variables.	-ZZZn = 0 (0 ≤ n ≤ 15)	As specified	p.448
-WAR[NING] = n (0 ≤ n ≤ 1FH)	Controls warning message output.	-WAR = 0	As specified	p.449
-INC[LUDE] = <path-name>	Specifies a search path where an include file is to be searched for.	—	As specified	p.450
-ABS[OLUTE]	Specifies the absolute mode.	No specification (relocatable mode)	As specified	p.451
-XRE[F][=<path-name>] -NOX[REF]	Controls intermediate cross-reference file output.	-XRE	As specified	p.452
-TAGS[TART] = "character string"	Specifies a tag start character string (valid only when -HOST is specified).	No specification	As specified	p.453
-TAGE[ND] = "character string"	Specifies a tag end character string (valid only when -HOST is specified).	No specification	As specified	p.454
-SUM[MARY] -NOS[UMMARY]	Controls summary file output.	-NOS	-SUM	p.455

—: Indicates that there is no influence or specification.

22.6.1 Object Output Control (-OBJ, -NOO)

[Format]

```
-OBJ [ECT] [=<path-name>]                                    (Default: -OBJ)
-NOO [BJECT]
```

[Function]

Controls object file (.REL) output.

[Explanation]

(1) -OBJ [ECT] [=<path-name>]
Specifies the path name where the object file is to be output.
If a path name is not specified with -OBJ, a default assumption described in item (3) is used.

(2) -NOO [BJECT]
Specifies that no object file be output.

(3) If this option is not specified (default)
The object code is output to a file having the same name as the source module file name (file name extension .REL) in the current directory.

[Relationships with other options]

(1) If -HOST is specified, the <path-name> specified in the form of -OBJ=<path-name> and a specification of -NOO become invalid, and -OBJ is assumed.

(2) A specification of -OBJ or -NOO does not affect other options.

[Notes]

(1) A file-name cannot be specified for an object file. If an attempt is made to specify one, an error (A106: Invalid option (option_name)) is detected, and processing is aborted.

(2) If an attempt is made to specify a nonexistent <path-name>, an error (A106: Invalid option (option_name)) is detected.

22.6.2 List Output Control (-LIS, -NOL)

[Format]

```
-LIS [T] [=<path-name> [\<file-name>]]                    (Default: -LIS)
-NOL [IST]
```

[Function]

Controls intermediate list file output.

[Explanation]

(1) -LIS [T]
If a specification of <path-name>\<file-name> is not provided, the intermediate list is output to a file having the same name as the source module file name (file name extension .IPN) in the current directory.

(2) -LIS [T]=<path-name>
The intermediate list is output to a file having the same name as the source module file name (file name extension .IPN) in a directory specified in <path-name>.

(3) -LIS [T]=<file-name>
The intermediate list is output to a file named <file-name> in the directory where the source module file is. If a file name extension is omitted, .IPN is assumed.

(4) -LIS [T]=<path-name>\<file-name>
The intermediate list is output to a file named <file-name> in a directory specified in <path-name>. If a file name extension is omitted, .IPN is assumed.

(5) -NOL [IST]
An intermediate list file is not output.

(6) If this option is not specified (default)
-LIS is assumed.

[Relationships with other options]

(1) If -HOST is specified, the <path-name> and <file-name> specified in the form of -LIS=<path-name> [\<file-name>], and a specification of -NOL become invalid, and -LIS is assumed.

(2) A specification of -LIS or -NOL does not affect other options.

[Notes]

(1) If an attempt is made to specify a nonexistent <path-name>, an error (A106: Invalid option (option_name)) is detected.

22.6.3 Undefined Symbol File Output Control (-UND, -NOU)

[Format]

```
-UND [EF] [=<path-name> [\<file-name>]]                    (Default: -NOU)
-NOU [NDEF]
```

[Function]

Controls undefined symbol file output.

[Explanation]

(1) -UND [EF]
If a specification of <path-name>\<file-name> is not provided, undefined symbols are output to a file having the same name as the source module file name (file name extension .UND) in the current directory.

(2) -UND [EF]=<path-name>
Undefined symbols are output to a file having the same name as the source module file name (file name extension .UND) in a directory specified in <path-name>.

(3) -UND [EF]=<file-name>
Undefined symbols are output to a file named <file-name> in the directory where the source module file is. If a file name extension is omitted, .UND is assumed.

(4) -UND [EF]=<path-name>\<file-name>
Undefined symbols are output to a file having the specified name in a directory specified in <path-name>. If a file name extension is omitted, .UND is assumed.

(5) -NOU [NDEF]
An undefined symbol is not output.

(6) If this option is not specified (default)
-NOU is assumed.

[Relationships with other options]

(1) A specification of -UND or -NOU does not affect other options.

[Notes]

(1) If an attempt is made to specify a nonexistent <path-name>, an error (A106: Invalid option (option_name)) is detected.

22.6.4 Work Drive Control (-WOR)

[Format]

```
-WOR[K]=<path-name>                                (Default: Not specified)
```

[Function]

Specifies <path-name> where a work file is to be reserved for assembly.

[Explanation]

(1) If only a drive name is specified in <path-name>, a work file is created in the current directory.

(2) If only a directory name is specified in <path-name>, a work file is created in the specified directory on the current drive.

(3) When this option is not specified, if <path-name> is specified in the TMP environment variable, the path name in <path-name> is handled as a work drive. If <path-name> is not specified in the environment variable, the current path is handled as a work drive.

[Relationships with other options]

(1) A specification of -WOR=<path-name> does not affect other options.

[Notes]

(1) If an attempt is made to specify a nonexistent <path-name>, an error (A106: Invalid option (option_name)) is detected.

(2) All work files are deleted after assembly.

* (3) 10 MB or more of free space should be available in the work drive.

22.6.5 *SIMPLEHOST* Information Control (-HOS, -NOH)

[Format]

```
-HOS [T]                                                    (Default: -NOH)
-NOH [OST]
```

[Function]

Controls output of information necessary in using *SIMPLEHOST*.

[Explanation]

(1) -HOS [T]
The *SIMPLEHOST* information is output to the .REL file.

(2) -NOH [OST]
The *SIMPLEHOST* information is not output.

(3) If this option is not specified (default)
-NOH is assumed.

[Relationships with other options]

(1) A specification of -NOH makes invalid the following options:
- -TAGS [TART]="character string"
- -TAGE [ND]="character string"

(2) A specification of -HOST causes default assumptions to be used for the following options:
- Object output control option -> -OBJ
- List output control option -> -LIS
- Summary file output control option -> -SUM

22.6.6 Assemble Time Variable (-ZZZn)

[Format]

```
-ZZZn=m   (0 ≤ n ≤ 15 where n is a decimal integer)  (Default: -ZZZn = 0)
          (0 ≤ m ≤ 0FFFFFFFFH)
```

[Function]

Initializes the ZZZn assemble time variable to value m.

[Relationships with other options]

(1) A specification of -ZZZn does not affect other options.

[Notes]

(1) Value m should evaluate to within a range between 0 and 0FFFFFFFFH. Otherwise, an error (invalid option) is detected, and assembly is aborted.

(2) Value m can be any of binary, decimal, and hexadecimal numbers. If an expression or character string is specified as m, an error (invalid option) is detected, and assembly is aborted.

(3) If this option is not specified when RA17K is started, the assemble time variable is initialized to 0.

22.6.7 Warning Output Level Control (-WAR)

[Format]

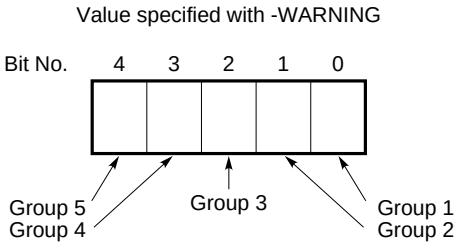
```
-WAR [NING]=n                    (n = 0 to 1FH) (Default: -WAR = 00000B)
```

[Function]

Specifies outputs of various warnings during assembly.

[Explanation]

Warning messages are grouped into six. A 5-bit value is specified with -WARNING. Each bit corresponds to a warning message group. Group 6 is always checked regardless of the warning level, however.



When a bit is 0:

The corresponding group is checked for a warning.

When a bit is 1:

The corresponding group is not checked for a warning.

See **Chapter 24** for the grouping of the warning messages.

[Relationships with other options]

(1) A specification of -WAR=n does not affect other options.

[Notes]

(1) If an attempt is made to specify n as a number that does not fall in a range between 0 and 1FH, an error (A106: Invalid option (option_name)) is detected. n can be any of binary, decimal and hexadecimal numbers.

[Example]

```
-WAR=10110B
```

Groups 5, 3, or 2 is not checked for a warning. Group 6 is always checked for a warning.

22.6.8 Include File Search Path Specification (-INC)

[Format]

```
-INC [LUDE]=<path-name> [;<path-name>;...]
```

[Function]

Specifies a path name where the desired include file exists.

[Explanation]

(1) More than one path name can be specified by separating them with a semicolon (;). If more than one path name is specified, the path names are searched for the target include file in the stated order.

(2) If the target include file is not in any <path-name> specified in this option, a <path-name> specified in the environment variable (INC17K) is searched through. If the target include file is not found in this <path-name> either, the directory holding the source module file is searched through.

[Relationships with other options]

(1) A specification of -INC=<path-name> [;<path-name>;...] does not affect other options.

[Notes]

(1) If a <path-name> specified in the form of -INC=<path-name> [;<path-name>;...] does not exist, an error is not detected.

22.6.9 Assemble Mode Control (-ABS)

[Format]

```
-ABS [OLUTE]                (Default: If not specified = relocatable mode)
```

[Function]

Specifies whether assembly be carried out in the relocatable or absolute mode.

[Explanation]

(1) When this option is specified, assembly is carried out in the absolute mode.

(2) When this option is not specified (default), assembly is carried out in the relocatable mode.

[Relationships with other options]

(1) A specification of -ABS does not affect other options.

[Notes]

(1) If a source module file contains a description which is not usable in each assemble mode, an error is detected (see **Chapter 2**).

22.6.10 Intermediate Cross-Reference Output Control (-XRE, -NOX)

[Format]

```
-XRE [F] [=<path-name> [\<file-name>]]                    (Default: -XRE)
-NOX [REF]
```

[Function]

Controls intermediate cross-reference output.

[Explanation]

(1) -XRE [F]
If a specification of <path-name>\<file-name> is not provided, intermediate cross-references are output to a file having the same name as the source module file name (file name extension .IXF) in the current directory.

(2) -XRE [F]=<path-name>
Intermediate cross-references are output to a file having the same name as the source module file name (file name extension .IXF) in a directory specified in <path-name>.

(3) -XRE [F]=<path-name>\<file-name>
Intermediate cross-references are output to a file named <file-name> in a directory specified in <path-name>. If a file name extension is omitted, .IXF is assumed.

(4) -NOX [REF]
An intermediate cross-reference list is not output.

(5) If this option is not specified (default)
-XRE is assumed.

[Relationships with other options]

(1) A specification of -XRE or -NOX does not affect other options.

[Notes]

(1) If an attempt is made to specify a nonexistent <path-name>, an error (A106: Invalid option (option_name)) is detected.

22.6.11 Tag Start Character String Specification (-TAGS)

[Format]

```
-TAGS [TART]="character string"                    (Default: Not specified)
```

[Function]

Replaces the tag start character string "." with another character string. The character string must be enclosed in double quotations.

[Explanation]

(1) -TAGS [TART]="character string"
The specified character string is used as the tag start character string.

(2) If this option is not specified (default)
A period "." is used as the tag start character.

[Relationships with other options]

(1) A specification of -TAGS="character string" does not affect other options.

(2) This option is valid only when -HOST is specified.

[Notes]

(1) The tag start character string can consist of up to 64 characters excluding double quotations. If an attempt is made to specify a larger character string, an error (A106: Invalid option (option_name)) is detected.

(2) If an attempt is made to specify a null character, an error (A106: Invalid option (option_name)) is detected.

22.6.12 Tag End Character String Specification (-TAGE)

[Format]

```
-TAGE [ND]="character string"                    (Default: Not specified)
```

[Function]

Replaces the tag end character string ".." with another character string. The character string must be enclosed in double quotations.

[Explanation]

(1) -TAGE [ND]="character string"
The specified character string is used as the tag end character string.

(2) If this option is not specified (default)
A pair of periods ".." is used as the tag end character string.

[Relationships with other options]

(1) A specification of -TAGE does not affect other options.

(2) This option is valid only when -HOST is specified.

[Notes]

(1) The tag end character string can consist of up to 64 characters excluding double quotations. If an attempt is made to specify a larger character string, an error (A106: Invalid option (option_name)) is detected.

(2) If an attempt is made to specify a null character, an error (A106: Invalid option (option_name)) is detected.

22.6.13 Summary File Output Control (-SUM, -NOS)

[Format]

```
-SUM [MARY]                                                    (Default: -NOS)
-NOS [UMMARY]
```

[Function]

Controls summary file output.

[Explanation]

(1) -SUM [MARY]
A summary file is created under the same name as the source module file name (file name extension .SUM) in the current directory.

(2) -NOS [UMMARY]
A summary file is not output.

(3) If this option is not specified (default)
-NOS is assumed.

[Relationships with other options]

(1) A specification of -SUM or -NOS does not affect other options.

(2) If -HOST is specified, -NOS becomes invalid, and -SUM is assumed.

[MEMO]

CHAPTER 23 OUTPUT LIST FORMATS

This chapter describes the formats of the intermediate list, log, and undefined symbol files created by RA17K.

23.1 INTERMEDIATE LIST FILE

RA17K outputs assembly results to an intermediate list file. It is possible to prohibit listing by using an option.

The intermediate list file is not affected by the list control pseudo instruction. Processing by the list control pseudo instruction takes effect, when the intermediate list file is converted to an absolute-address-based list file using the document processor (DOC17K).

Object code is in a fixed format of 4-4-4-4 bit arrangement (four hexadecimal digits).

The format of each intermediate list line is as follows:

```
EEEE SSSSSS LLLLLL OOOOOOOO MM I CL      sssssss.....
```

Symbol	Start position	Number of digits	Description	Reference page
E	Column 1	4	Error/EPA field	p.459
S	Column 6	6	Source line number field	p.460
L	Column 13	6	Location counter field	p.462
O	Column 20	8	Object code field	p.463
M	Column 29	2	Macro nest field	p.465
I	Column 32	1	Include nest field	p.466
C	Column 34	1	Control field	p.467
L	Column 35	1	Label field	p.470
s	Column 41	Variable	Source field	p.470

Each field is explained in the following sections.

23.1.1 Error/EPA Field

[Explanation]

(1) This field holds an error code, if an error occurs.

(2) If the EPA bit is 1, "1" is placed in this field, left-justified.

(3) If an error code is to be output, and the EPA bit becomes 1 simultaneously, error code output takes precedence.

(4) If the EPA bit is 0, and no error has occurred, this field is set with four space characters.

[Output example]

(1) Error code

```
EEEE    SSSSSS    LLLLLL···
F011
```

(2) EPA bit

```
EEEE    SSSSSS    LLLLLL···
1
```

"1" is left-justified in the field, and the remaining positions (3 positions) in the field are set with space characters.

23.1.2 Source Line Number Field

[Explanation]

(1) This field holds a decimal representation of a line number in the source module file.

(2) When include files or macros are expanded, their line numbers are preceded by the sign "+".

(3) For expansions of include files and macros (including built-in macros and repetitive pseudo instructions), the line number begins at 1 in a nest one level down the normal line number level. Even if further nesting occurs, the line numbers for the include file and macro expansions are kept consecutive no matter how many nesting levels are there.

(4) The line number begins at 1, and can be as high as 65535. If there are more than 65535 source lines, an error (A165: Source line over) is detected, and assembly is aborted. This holds true also of the line numbers of include file and macro expansions.

(5) The line number is right-justified in the source line number field. For include file and macro expansion line numbers, the sign "+" is placed in the leftmost position in the field.

[Output example]

(1) Ordinary source line number output

```
EEEE      SSSSSS     LLLLLL   ...

              :
            123
            124
            125
              :
```

(2) Include file expansion line numbers

```
EEEE      SSSSSS     LLLLLL   ...    SSSS...

                :
               123
               124                    INCLUDE ' ABC.ASM'
             +   1   ┐ INCLUDE expansion line numbers
             +   2   │
             +   3   ┘
               125
                :
```

(3) Macro expansion line numbers

```
EEEE      SSSSSS     LLLLLL   ...    SSSS...

                :
               123                    ABC  <- Macro reference
             +   1   ┐ Macro expansion line numbers
             +   2   ┘
               124
                :
```

23.1.3 Location Counter Field

[Explanation]

(1) If an instruction in a source line is one that causes an object code to be generated or one that evaluates to a location counter value, the location counter value corresponding to the line is output to this field.

(2) The numbers output to this field are hexadecimal numbers, and represented using uppercase characters. The location counter value is 5 digits, and right-justified in the field. If the value is smaller than 5 digits, leading zeros are inserted to make it 5 digits.

(3) A location counter value is either an offset address (relative address) or an absolute address. The absolute address is identified with "#" placed in the leftmost position in the 6-digit field.

[Output example]

(1) Relative address

```
···SSSSSS     LLLLLL···

               01000
               01001
               01002
```

(2) Absolute address

```
···SSSSSS     LLLLLL···

              #01000
              #01001
              #01002
```

23.1.4 Object Code Field

[Explanation]

(1) The object code field indicates the object code of an instruction or pseudo instruction, or a value to which a symbol definition pseudo instruction evaluates.

Pseudo instructions generating object code

- DB
- DW
- DCP

Pseudo instructions that evaluate to a value

- DAT
- MEM
- NIBBLEn
- NIBBLEnV
- FLG
- LAB
- SET
- Mask option pseudo instruction
- ZZZMCHK

(2) An object code is output in 4-4-4-4 bit format (4-digit hexadecimal number), and right-justified in the field with space placed in the left-side four positions.

(3) An evaluated value is output as an 8-digit hexadecimal number. If it is smaller than 8 digits, leading zeros are inserted to make it 8 digits.

(4) If an object code is not generated when it should be (for example, if an operand is an externally defined symbol), the object code field is set with " ****".

[Output example]

(1) Object code

```
···LLLLLL    00000000   ···    SSSSS···

                 3CF0          NOP
                 3CF0          NOP
```

(2) Evaluated value

```
···LLLLLL    00000000   ···    SSSSS···

             00001234          AAA DAT 1234H
             00000123          BBB MEM 1. 23H
```

(3) If an object code is not generated when it should be

```
···LLLLLL    00000000   ···    SSSSS···

                 ****          ADD MEM00, #1H
```

23.1.5 Macro Nest Field

[Explanation]

(1) This field indicates the nesting level of a macro, repetitive pseudo instruction, or built-in macro.

(2) For macros, the nesting level is increased by 1 on the macro reference line (or, for a built-in macro, on the line where it is described).

(3) For repetitive pseudo instructions, the nesting level is output to the expansion line.

[Output example]

(1) Macro nesting level

```
···MM      I    CL      SSSSS···

                        ABC    MACRO
                               NOP
                               ENDM
     1                         ABC
     1                         NOP
```

(2) Repetitive pseudo instruction nesting level

```
···MM      I    CL      SSSSS···

                        REPT   2
                        NOP
                        ENDR
     1                  NOP
     1                  NOP
```

(3) Nesting level of a repetitive pseudo instruction in a macro body

```
···MM      I    CL      SSSSS···

                        ABC    MACRO
                        NOP
                        REPT   2
                        INC    IX
                        ENDR
                        NOP
                        ENDM
      1                 ABC
      1                 NOP
      2                 INC    IX
      2                 INC    IX
      1                 NOP
```

23.1.6 Include Nest Field

[Explanation]

(1) This field indicates a nesting level from 1 to 8.

(2) The nesting level is incremented by 1 on the line where the INCLUDE statement is described.

[Output example]

```
···MM      I    CL      SSSSS···

                            ⋮
           1                INCLUDE   ' ABCD. ASM'
           1                    ⋮
           1                    ⋮
           2                INCLUDE   ' BBB. ASM'  ; Nested INCLUDE statement
           2                    ⋮
           2                    ⋮
```

23.1.7 Control Field

[Explanation]

This field indicates a control code that briefs the corresponding source line for the document processor (DOC17K).

Control codes output to the control field (1/3)

Control code	Description	
A	Line where a symbol definition pseudo instruction is described	DAT, LAB, MEM, FLG, SET, NIBBLE NIBBLEn, NIBBLEnV, ZZZMCHK
D	Line where a data definition pseudo instruction is described	DW, DB, DCP
M	Line where a macro definition pseudo instruction is described	MACRO, REPT, IRP
m	Line where a macro definition ends	ENDM, ENDR
=	Line where an EXITR is described	
G	Line where an ORG is described	
H	Line where an absolute-mode CSEG is described	
X	Line where a relocatable-mode CSEG is described	
T	Line where a relocatable-mode CSEG is described (when TABLE is specified)	
Z	Line where an ENSURE is described	
P	Line where a PUBLIC is described	
*	Line where a PUBLIC BELOW is described	
+	Line where an ENDP is described	
E	Line where an EXTRN is described	
F	Line where a LITERAL or UNLITERAL is described	
I	Line where an INCLUDE is described	
J	All lines to be skipped during conditional assembly IF, ELSE, ENDIF, CASE, EXIT, ENDCASE, IFCHAR, ENDIFC IFNCHAR, ENDIFNC, IFSTR, ENDIFS	
C	Line where a CASE is described	
c	Line where an ENDCASE is described	
;	Comment line (line containing only a comment or nothing), SUMMARY, and tag summary statement lines	
U	Line referencing a user-defined macro (macro other than built-in macros)	
z	Line referencing a mask option macro	
R	Line where a SUMMARY is described	
r	Line where SUMMARY ends (ENDSUM line or line where a termination symbol is described)	
.	Line where a TAG is described (;. and ;.V)	

Control codes output to the control field **(2/3)**

Control code	Description
/	Line where a TAG is described (;..)
Q	Line where an OPTION is described
q	Line where an ENDOP is described
0	Line where a NOLIST is described
1	Line where a LIST is described
3	Line where a C14344 is described
4	Line where a C4444 is described
5	Line where a TITLE is described
6	Line where an EJECT is described
7	Line where an SFCOND is described
8	Line where an LFCOND is described
S	Line where an SMAC is described
N	Line where a NOMAC is described
O	Line where an OMAC is described
L	Line where an LMAC is described
V	Line where a VMAC is described
s	Line where an SBMAC is described
n	Line where a NOBMAC is described
o	Line where an OBMAC is described
l	Line where an LBMAC is described
v	Line where a VBMAC is described
a	END
b	EOF
\	ZZZERROR
d	ZZZMSG
e	GLOBAL
f	PURGE
g	Line where a BR instruction is described
h	Line where a CALL instruction is described
i	Line where a SYSCAL instruction is described
j	Line where a RET instruction is described
k	Line where a RETSK instruction is described
p	Line where a RETI instruction is described

Control codes output to the control field **(3/3)**

Control code	Description
!	Mnemonic group 1 (reference-only instructions) SKE SKGE SKLT SKNE SKT SKF
"	Mnemonic group 2 (instructions with no operand) DI EI NOP
#	Mnemonic group 3 (instructions with one operand) INC PUSH POP RORC STOP HALT
$	Mnemonic group 3 (instructions with two operands) ADD ADDC SUB SUBC AND OR XOR LD ST MOV MOVTH MOVTL PEEK POKE GET PUT
y	Structured instruction
%	Built-in macro group 1 (reference-only instructions) SETBANK SETRP SETMP SETIX SETAR SKTn SKFn SKTX SKFX SKEX SKNEX SKGEX SKGTX SKLEX SKLTX BRX CALLX MOVTX SYSCALX
&	Built-in macro group 2 (instructions with no operand) BANKn
—	Built-in macro group 3 (the other instructions) SETn CLRn NOTn INITFLG SETX CLRX NOTX MOVX ADDX ADDCX ADDSX ADDCSX SUBX SUBCX SUBSX SUBCSX RORCX ROLCX SHRX SHLX ANDX ORX XORX INITFLGX

[Output example]

For control instructions that appear during an assemble skip, the J control code indicating that a skip is in progress is output.

```
    DDD DAT 1
    IF  DDD=1
        NOP
        NOP
    ELSE
J       NOP
J       NOP
    ENDIF

    CASE    DDD
J   0:
J       NOP
J       NOP
    1:
        NOP
        NOP
    OTHER:
        NOP
        NOP
    ENDCASE

    CASE    DDD
J   0:
J       NOP
    1:
        NOP
        NOP
        EXIT
J       NOP
    ENDCASE
```

23.1.8 Label Field

[Explanation]

For a line where a label is described, the label field indicates letter L. For a line where no label is described, the label field is set with " ".

23.1.9 Source Field

[Explanation]

The source field contains exactly what the user describes in a source file. A TAB is not converted to a space, or vice versa.

23.2 LOG FILE

[Explanation]

(1) The log file holds all data displayed on the screen during assembly.

(2) The log file is created automatically when RA17K runs. Its file name is in the form: Source-module-file-name.LOG

(3) If assembly is aborted because of an error before the source module file name is specified, the log file is named: RA17K.LOG

The following example shows the contents of a log file created when TEST1.ASM is assembled.

```
17K Series Relocatable Assembler V1.xx [DD MMM YY]
    Copyright(C)NEC Corporation 19xx

TEST1.ASM assemble start hh:mm:ss MM/DD/YY

DB  P1
TEST1.ASM(9) error F058: Undefined symbol
    DB      P1
TEST1.ASM(10) error F058: Undefined symbol

MEM AREA  = REL       TMP   :   4317 :    259
MEM AREA  = REL       TMP   :   4317 :    229
MEM AREA  = REL       TMP   :   4317 :   4317
FILE NAME = E:\r$b05938 :  12000
MEM AREA  = REL       TMP   :   4317 :   4317
FILE NAME = E:\r$a05938 :  78000
MEM AREA  = REL       TMP   :   4317 :    225
MEM AREA  = REL       TMP   :   2269 :   2269
FILE NAME = E:\r$c05938 :  120440
MEM AREA  = REL       TMP   :   4317 :    260
MEM AREA  = REL       FILE  :   4321 :   4321
FILE NAME = TEST1.REL   :180656
MEM AREA  = INTERM    TMP   :   8413 :   8413
FILE NAME = E:\r$005938 :  159021
MEM AREA  = XREF      FILE  :   8417 :   8417
FILE NAME = TEST1.IXF :   48000
MEM AREA  = SYMBOL    MAIN  :  46592 :  46592
MEM AREA  = SYMBOL    EMS   : 114688 : 114688
MEM AREA  = LIST      TMP   :   4321 :   4321
FILE NAME = TEST1.IPN :  443306
MEM AREA  = ETC             :5223    :    404

Total error(s): 2 Total warning(s): 0
TEST1.ASM assemble end hh:mm:ss MM/DD/YY
```

In this example, two error have occurred, but a warning has not.

23.3 UNDEFINED SYMBOL FILE

[Explanation]

(1) The undefined symbol file contain symbols detected as undefined during assembly.

(2) If the -NOUNDEF option is specified, an undefined symbol file is not created even when an undefined symbol is found.

(3) If the -UNDEF option is specified, an undefined symbol file is created even when no undefined symbol is found.

(4) Because -NOUNDEF is a default assumption, an undefined symbol file is not created by default.

An example of undefined symbol file output follows.

[Output example]
In the following example, symbols FLG1, FLG2, FLG3, and FLG4 have been detected as undefined.

```
EXTRN ???:FLG1
EXTRN ???:FLG2
EXTRN ???:FLG3
EXTRN ???:FLG4
```

CHAPTER 24 RA17K ERROR MESSAGES

24.1 MESSAGES

The error and warning messages which may be output by RA17K are listed below. The first letter of each message number indicates the message type: error, warning, or abort.

First letter

F : Error message
W : Warning message
A : Abort message

[Warning groups]

Warning messages are classified into six groups (the number enclosed in parentheses under the number of each warning message indicates the warning group number of that message). The user can control whether warning messages are to be output, in group units, using the -WARNING option. Messages in warning group 6 are, however, always output if the corresponding condition occurs, regardless of the setting of the -WARNING option (see **Section 22.6.7**).

Message No.	Description	
F011	Message	Illegal first operand type
	Cause	The type of the first operand for a code-generating instruction is invalid. • Operation is performed between different types. • A type not allowed for the operand is described.
	Action	Describe an expression of a valid type.
F012	Message	Illegal second operand type
	Cause	The type of the second operand for a code-generating instruction is invalid. • Operation is performed between different types. • A type not allowed for the operand is described.
	Action	Describe an expression of a valid type.
F014	Message	Illegal first operand value
	Cause	The evaluated value of the first operand for a code-generating instruction falls outside the valid range.
	Action	Describe the operand such that the evaluated value falls within the valid range.
F015	Message	Illegal second operand value
	Cause	The evaluated value of the second operand for a code-generating instruction falls outside the valid range.
	Action	Describe the operand such that the evaluated value falls within the valid range.

Message No.	Description	
W020 (1)	Message	Unreference symbol (DAT type/MEM type/FLG type)
	Cause	A symbol is defined but not referenced.
	Action	Delete the symbol if not necessary.
W021 (5)	Message	Unreference symbol (LAB type)
	Cause	A symbol is defined but not referenced.
	Action	Delete the symbol if not necessary.
W022 (1)	Message	The SUMMARY statement described before is invalid
	Cause	Two or more summary statements are described before the CSEG statement.
	Action	Delete unnecessary summary statements.
W028 (2)	Message	No END statement
	Cause	A source module file is read in its entirety, but no END statement is found.
	Action	Describe an END statement at the end of the source module file.
F029	Message	No ENDIF statement
	Cause	A source module file is read in its entirety, but an ENDIF statement corresponding to an IF statement cannot be found.
	Action	Insert an ENDIF statement where appropriate.
F030	Message	No ENDCASE statement
	Cause	A source module file is read in its entirety, but an ENDCASE statement corresponding to a CASE statement cannot be found.
	Action	Insert an ENDCASE statement where appropriate.
F031	Message	No ENDR statement
	Cause	A source module file is read in its entirety, but an ENDR statement corresponding to an IRP or REPT statement cannot be found.
	Action	Insert an ENDR statement where appropriate.
F032	Message	No ENDM statement
	Cause	A source module file is read in its entirety, but an ENDM statement corresponding to a MACRO statement cannot be found.
	Action	Insert an ENDM statement where appropriate.
F033	Message	No ENDP statement
	Cause	A source module file is read in its entirety, but an ENDP statement corresponding to a PUBLIC BELOW statement cannot be found.
	Action	Insert an ENDP statement where appropriate.
F034	Message	No ENDOP statement
	Cause	A source module file is read in its entirety, but an ENDOP statement corresponding to an OPTION statement cannot be found.
	Action	Insert an ENDOP statement where appropriate.

Message No.	Description	
A035	Message	Nesting overflow
	Cause	• The nesting level for MACRO, IRP, or REPT exceeds the maximum (40). • The nesting level for IF, CASE, IFCHAR, IFNCHAR, or IFSTR exceeds the maximum (40).
	Action	Reduce the nesting level to 40 or less.
F036	Message	Operand count error
	Cause	The number of operands is invalid.
	Action	Describe a valid number of operands.
F037	Message	Syntax error
	Cause	• The syntax is erroneous. • A label is described where no label can be described. • An operand is described where no operand can be described.
	Action	Correct the syntax.
A038	Message	System memory overflow
	Cause	Memory to be used by the system is insufficient.
	Action	Increase the memory capacity. Alternatively, split the source module file into several small files.
A039	Message	Symbol or macro area overflow
	Cause	The area for storing symbols and macro bodies is insufficient.
	Action	Increase the memory capacity. Alternatively, split the source module file into several small files.
F040	Message	Invalid EOF statement
	Cause	An EOF statement is found in a file other than an include file.
	Action	Delete the EOF statement if not necessary.
F041	Message	Invalid ENDR statement
	Cause	An ENDR statement is found although there is no corresponding IRP or REPT statement.
	Action	Insert an IRP or REPT statement where appropriate. Alternatively, delete the ENDR statement if not necessary.
F042	Message	Invalid EXITR statement
	Cause	An EXITR statement is found outside the range between IRP and ENDR or between REPT and ENDR.
	Action	Move the EXITR statement to an appropriate position or delete it if not necessary.
F043	Message	Invalid ENDM statement
	Cause	An ENDM statement is found although there is no corresponding MACRO statement.
	Action	Insert a MACRO statement where appropriate. Alternatively, delete the ENDM statement if not necessary.

Message No.	Description	
F044	Message	Invalid value
	Cause	• The evaluated value of an expression described in a pseudo instruction (or the like) falls outside the valid range. • Segment numbers for a CSEG pseudo instruction are not described in ascending order. • An uninstalled segment number is described for a CSEG pseudo instruction. • An invalid numerical constant is described.
	Action	Correct the erroneous description.
F045	Message	Invalid type
	Cause	The type of an operand specified for a pseudo instruction (or the like) is invalid. • Operation is performed between different types. • A type not allowed for the operand is described. • For a type conversion function, the actual type of the expression differs from the current type specified for that function.
	Action	Describe an expression of a valid type.
F046	Message	Invalid BANK No.
	Cause	The specified bank number does not exist in the target device.
	Action	Specify a valid bank number.
F048	Message	ORG address error
	Cause	The specified value is smaller than the current value of the location counter.
	Action	Specify a valid value.
F049	Message	Used reserved word
	Cause	The string specified for symbol definition is a reserved word.
	Action	Specify a string other than reserved words.
F050	Message	Not reserved word
	Cause	Characters other than valid reserved words are described.
	Action	Specify a valid reserved word.
F051	Message	Invalid data length
	Cause	The specified number of characters exceeds the specified value.
	Action	Specify a valid data length.
F052	Message	Include nesting error
	Cause	The nesting level for include files exceeds the maximum (8).
	Action	Reduce the level to eight or smaller.
F053	Message	Duplicated OPTION directive
	Cause	An OPTION pseudo instruction is defined twice.
	Action	Delete either definition which is not necessary.

Message No.	Description	
F057	Message	Symbol multi defined
	Cause	An attempt is made to define an already defined symbol.
	Action	Change the name of the symbol.
F058	Message	Undefined symbol
	Cause	An undefined symbol is used.
	Action	Use a defined symbol. Alternatively, define the symbol to be used.
F060	Message	Invalid mnemonic
	Cause	The described string is not an instruction.
	Action	Correct the string to a valid instruction or delete it if not necessary.
F061	Message	No include file <filename>
	Cause	The specified include file does not exist.
	Action	Specify an existing include file.
F063	Message	Bank unmatch
	Cause	The bank numbers specified for a symbol are not the same.
	Action	Specify the same bank numbers within an instruction.
W065 (2)	Message	Statement after END
	Cause	A statement follows an END statement.
	Action	Delete the statement if not necessary.
W066 (2)	Message	Statement after EOF
	Cause	A statement follows an EOF statement.
	Action	Delete the statement if not necessary.
F067	Message	Address error
	Cause	The specified data memory address is not installed on the target device.
	Action	Specify a valid address.
W068 (6)	Message	Operation in OPTION block
	Cause	A mask option definition block contains an instruction.
	Action	Delete the instruction.
F069	Message	Invalid CASE LABEL
	Cause	• A statement other than comments is described for a numerical label. • A numerical label is described without a CASE statement.
	Action	Correct or delete the relevant description, as required.
F070	Message	Invalid operand
	Cause	The operand field contains a syntax error.
	Action	Correct the syntax.

Message No.	Description	
F080	Message	Invalid EXIT statement
	Cause	An EXIT statement is described outside the range between CASE and ENDCASE.
	Action	Move the EXIT statement to an appropriate position or delete it if not necessary.
F081	Message	Boundary error
	Cause	• A DCP instruction is described such that an object code will be generated at an address the four low-order bits of which are 0FH. • A table block spans the program area and EPA area.
	Action	• Describe a DCP instruction, using ORG, such that an object code will be generated at an address the four low-order bits of which are other than 0FH. • Adjust the location and size of the table block such that it is contained in the program area.
F082	Message	Illegal character
	Cause	An illegal character is described.
	Action	Correct the character.
F083	Message	Illegal format
	Cause	An operand for a DCP instruction is described in illegal format.
	Action	Describe the operand in legal format.
F085	Message	Invalid ENDP statement
	Cause	An ENDP statement is described when no PUBLIC BELOW statement is defined.
	Action	Define a PUBLIC BELOW statement or delete the ENDP statement if not necessary.
F087	Message	Invalid OPTION statement
	Cause	An OPTION statement is described where it is not allowed.
	Action	Move the OPTION statement to an appropriate position or delete it if not necessary.
F088	Message	Mask-option multi defined
	Cause	The same mask option is defined twice.
	Action	Delete either definition which is not necessary.
F089	Message	Undefined mask-option
	Cause	A mask option has yet to be defined.
	Action	Define all mask options properly.

Message No.	Description	
W094 (6)	Message	Indirect addressing instructions may not work properly due to the program exceeded to EPA area
	Cause	The EPA area is used because the program is too large to be contained in the program area. It, therefore, cannot be determined whether the branch destination for an indirect addressing instruction falls within the program area or EPA area.
	Action	Reduce the program size or replace the indirect addressing instruction with another instruction.
A100	Message	System error xxx
	Cause	Internal error (xxx: internal number)
	Action	Contact NEC or an NEC agency.
F101	Message	No ENDIFC statement
	Cause	A source module file is read in its entirety, but an ENDIFC statement corresponding to an IFCHAR statement cannot be found.
	Action	Insert an ENDIFC statement where appropriate.
F102	Message	No ENDIFNC statement
	Cause	A source module file is read in its entirety, but an ENDIFNC statement corresponding to an IFNCHAR statement cannot be found.
	Action	Insert an ENDIFNC statement where appropriate.
F103	Message	No ENDIFS statement
	Cause	A source module file is read in its entirety, but an ENDIFS statement corresponding to an IFSTR statement cannot be found.
	Action	Insert an ENDIFS statement where appropriate.
A104	Message	Can't open file <filename>
	Cause	File <filename> is not found.
	Action	Specify a valid file name.
A105	Message	<filename> is not device file
	Cause	The file specified with the first parameter, at activation, is other than device files.
	Action	Specify a valid device file name.
A106	Message	Invalid option (option_name)
	Cause	Option option_name is invalid.
	Action	Specify a valid option.
F107	Message	Used extended reserved word
	Cause	A reserved word for expansion is used.
	Action	Change it to another symbol.

Message No.	Description	
A108	Message	<filename> File I/O error at writing
	Cause	An error occurred while writing a file. • The free disk space is insufficient. • The file is write-protected. • The drive is not ready.
	Action	Check the drive. If it is normal, increase the free disk space.
A109	Message	<filename> File I/O error at reading
	Cause	An error occurred while reading a file. • The disk contains a defective sector. • The file is read-protected. • The drive is not ready.
	Action	Check the drive.
W110 (6)	Message	Invalid mnemonic at the last of program
	Cause	• The last instruction in a section is other than a branch instruction and data definition pseudo instruction. • The instruction immediately before an ORG pseudo instruction changing the location address is other than a branch instruction or data definition pseudo instruction. • The instruction immediately before a CSEG pseudo instruction is other than a branch instruction or data definition pseudo instruction.
	Action	Describe a valid instruction.
W111 (4)	Message	The result is over 16 bits
	Cause	The operation result exceeds 16 bits in absolute mode.
	Action	Specify the operation such that the result becomes within 16 bits.
F112	Message	Impossible to use on relocatable mode
	Cause	The specified pseudo instruction cannot be used in relocatable mode.
	Action	Delete the pseudo instruction.
F113	Message	Impossible to use on absolute mode
	Cause	The specified pseudo instruction cannot be used in absolute mode.
	Action	Delete the pseudo instruction.
W114 (6)	Message	The address carried out an operation using $ may be incorrect
	Cause	The address cannot be determined because internal processing is performed in relocatable mode.
	Action	Replace $ with a label.
W115 (6)	Message	ZZZn is not initialized
	Cause	ZZZn is used without initializing it. (This message is output only in absolute mode.)
	Action	Initialize ZZZn.

Message No.	Description	
F120	Message	Invalid ELSE statement
	Cause	An ELSE statement is found although there is no corresponding IF, IFCHAR, IFNCHAR, or IFSTR statement.
	Action	Insert an IF, IFCHAR, IFNCHAR, or IFSTR statement where appropriate. Alternatively, delete the ELSE statement if not necessary.
F121	Message	Invalid ENDIF statement
	Cause	An ENDIF statement is found although there is no corresponding IF statement.
	Action	Insert an IF statement where appropriate. Alternatively, delete the ENDIF statement if not necessary.
F122	Message	Invalid ENDCASE statement
	Cause	An ENDCASE statement is found although there is no corresponding CASE statement.
	Action	Insert a CASE statement where appropriate. Alternatively, delete the ENDCASE statement if not necessary.
F123	Message	Invalid OTHER statement
	Cause	An OTHER statement is described outside the range between CASE and ENDCASE.
	Action	Move the OTHER statement to an appropriate position or delete it if not necessary.
F124	Message	Invalid ENDIFC statement
	Cause	An ENDIFC statement is found although there is no corresponding IFCHAR statement.
	Action	Insert an IFCHAR statement where appropriate. Alternatively, delete the ENDIFC statement if not necessary.
F125	Message	Invalid ENDIFNC statement
	Cause	An ENDIFNC statement is found although there is no corresponding IFNCHAR statement.
	Action	Insert an IFNCHAR statement where appropriate. Alternatively, delete the ENDIFNC statement if not necessary.
F126	Message	Invalid ENDIFS statement
	Cause	An ENDIFS statement is found although there is no corresponding IFSTR statement.
	Action	Insert an IFSTR statement where appropriate. Alternatively, delete the ENDIFS statement if not necessary.
F127	Message	No _ENDIF statement
	Cause	A source module file is read in its entirety, but an _ENDIF statement corresponding to an _IF statement cannot be found.
	Action	Insert an _ENDIF statement where appropriate.

Message No.	Description	
F128	Message	Invalid _ELSEIF statement
	Cause	• An _ELSEIF statement is found although there is no corresponding _IF statement. • An _ELSEIF statement is duplicately specified. • An _ELSEIF statement is described after an _ELSE statement. • An _ELSEIF statement is described outside the range between _IF and _ENDIF.
	Action	Correct the description of the _ELSEIF statement or delete it if not necessary.
F129	Message	Invalid _ELSE statement
	Cause	• An _ELSE statement is found although there is no corresponding _IF statement. • An _ELSE statement is duplicately specified. • An _ELSE statement is described outside the range between _IF and _ENDIF.
	Action	Correct the description of the _ELSE statement or delete it if not necessary.
F130	Message	Invalid _ENDIF statement
	Cause	An _ENDIF statement is found although there is no corresponding _IF statement.
	Action	Insert an _IF statement where appropriate. Alternatively, delete the _ENDIF statement if not necessary.
F131	Message	No _ENDW statement
	Cause	A source module file is read in its entirety, but an _ENDW statement corresponding to a _WHILE statement cannot be found.
	Action	Insert an _ENDW statement where appropriate.
F132	Message	Invalid _ENDW statement
	Cause	An _ENDW statement is found although there is no corresponding _WHILE statement.
	Action	Insert an _WHILE statement where appropriate. Alternatively, delete the _ENDW statement if not necessary.
F133	Message	Invalid _CASE statement
	Cause	• A _CASE statement is described outside the range between _SWITCH and _ENDS. • The same constant is specified for two or more _CASE statements.
	Action	Move the _CASE statement to an appropriate position or delete it if not necessary.
F134	Message	Invalid _DEFAULT statement
	Cause	• A _DEFAULT statement is described outside the range between _SWITCH and _ENDS.
	Action	Move the _DEFAULT statement to an appropriate position or delete it if not necessary.

Message No.	Description	
F135	Message	Invalid _ENDS statement
	Cause	An _ENDS statement is found although there is no corresponding _SWITCH statement.
	Action	Insert an _SWITCH statement where appropriate. Alternatively, delete the _ENDS statement if not necessary.
F136	Message	No _ENDS statement
	Cause	A source module file is read in its entirety, but an _ENDS statement corresponding to a _SWITCH statement cannot be found.
	Action	Insert an _ENDS statement where appropriate.
F137	Message	Invalid _UNTIL statement
	Cause	An _UNTIL statement is found although there is no corresponding _REPEAT statement.
	Action	Insert a _REPEAT statement where appropriate. Alternatively, delete the _UNTIL statement if not necessary.
F138	Message	No _UNTIL statement
	Cause	A source module file is read in its entirety, but an _UNTIL statement corresponding to a _REPEAT statement cannot be found.
	Action	Insert an _UNTIL statement where appropriate.
F139	Message	Invalid _NEXT statement
	Cause	A _NEXT statement is found although there is no corresponding _FOR statement.
	Action	Insert a _FOR statement where appropriate. Alternatively, delete the _NEXT statement if not necessary.
F140	Message	No _NEXT statement
	Cause	A source module file is read in its entirety, but a _NEXT statement corresponding to a _FOR statement cannot be found.
	Action	Insert a _NEXT statement where appropriate.
F141	Message	Invalid _BREAK statement
	Cause	A _BREAK statement is described outside the range between _WHILE and _ENDW, _REPEAT and _UNTIL, _FOR and _NEXT, or _SWITCH and _ENDS.
	Action	Move the _BREAK statement to an appropriate position or delete it if not necessary.
F142	Message	Invalid _CONTINUE statement
	Cause	A _CONTINUE statement is described outside the range between _WHILE and _ENDW, _REPEAT and _UNTIL, or _FOR and _NEXT.
	Action	Move the _CONTINUE statement to an appropriate position or delete it if not necessary.

Message No.	Description	
F143	Message	Invalid _GOTO statement
	Cause	A _GOTO statement is described outside the range of structured description.
	Action	Move the _GOTO statement to an appropriate position or delete it if not necessary.
F144	Message	Invalid ENDOP statement
	Cause	An ENDOP statement is found although there is no corresponding OPTION statement.
	Action	Insert an OPTION statement where appropriate. Alternatively, delete the ENDOP statement if not necessary.
F145	Message	Impossible to use out of macro
	Cause	An instruction which can be used only within a macro body is used outside a macro body.
	Action	Correct the instruction.
F146	Message	Impossible to write out of section block
	Cause	An instruction which generates an object code is used outside a section block.
	Action	Move the instruction into a section.
F147	Message	BOOT or INT multi defined
*	Cause	BOOT or INT is defined twice in ENTRY.
	Action	Delete the unnecessary definition.
F148	Message	Macro body is over 64 K bytes
*	Cause	A macro body exceeds a maximum size of 64K bytes.
	Action	Reduce the size of the macro body to 64K bytes or less.
W149 (3)	Message	Carried out sub-routine call for the label not defined ENTRY
*	Cause	A CALL instruction is executed for a label that is not defined in ENTRY.
	Action	Define the label in ENTRY.
F150	Message	Impossible to write the external symbol
	Cause	An externally defined symbol is described in an operand of a pseudo instruction (or the like).
	Action	Define another symbol and use that symbol instead of the externally defined symbol.
W151 (4)	Message	The address is in remain area
	Cause	An address within the REMAIN area is specified as a ROM address.
	Action	Reduce the program size.
F152	Message	The address is out of ROM
	Cause	An address within an area which cannot be controlled by the program counter on the target device (other than the EPA area) is specified as a ROM address.
	Action	Specify a valid address.

Message No.	Description	
W153 (6)	Message	The address is in EPA area
	Cause	An address within the EPA area is specified as a ROM address.
	Action	Reduce the program size.
W157 (2)	Message	Letters in a line are over 255
	Cause	More than 255 characters are described within a line.
	Action	Reduce the number of characters to 255 or less.
A158	Message	Parameter error at start time
	Cause	The format of a parameter specified at activation is invalid.
	Action	Specify a valid parameter.
A159	Message	Can't open temporary file
	Cause	The free disk space is insufficient.
	Action	Increase the free disk space.
A160	Message	Temporary file I/O error at reading
	Cause	The specified file is not found.
	Action	Check the disk.
A161	Message	Temporary file I/O error at writing
	Cause	• The free disk space is insufficient. • The drive is not ready.
	Action	Check the disk. If it is normal, increase the free disk space.
A162	Message	No ENDSUM statement
	Cause	An ENDSUM statement corresponding to a SUMMARY statement cannot be found.
	Action	Insert an ENDSUM statement where appropriate.
A163	Message	<filename> is not a source file
	Cause	An extension which cannot be specified is specified.
	Action	Specify a source module file.
F164	Message	The constant is over 32 bits
	Cause	A numerical constant has a value exceeding 32 bits.
	Action	Reduce the value of the numerical constant to 32 bits or less.
A165	Message	Source line over
	Cause	The number of lines in a source module file, or the number of lines of an expanded macro or include file exceeds the maximum value.
	Action	Split the source module file, macro, or include file.
F166	Message	Invalid EXTRN statement
	Cause	A reserved word (such as a mnemonic, function, assemble-time variable) is declared with EXTRN.
	Action	Delete any invalid symbols.

Message No.	Description	
F167	Message	Invalid PUBLIC statement
	Cause	A reserved word (such as a mnemonic, function, assemble-time variable) is declared with PUBLIC.
	Action	Delete any invalid symbols.
F168	Message	Nibble number unmatch
	Cause	The number of nibbles specified in an operand is larger than that specified with n in DATn, NIBBLEn, or NIBBLEnV.
	Action	Match the number of nibbles.
W169 (4)	Message	Omitted a surplus due to an input value is over a regular value
	Cause	A value specified in a DW or DB instruction exceeds one word or one byte, respectively. The value is rounded off.
	Action	Specify a value within a specified range.
F170	Message	Impossible to include a source file
	Cause	In a source file, that file itself is specified as an include file.
	Action	Specify a valid include file.
F172	Message	Invalid CSEG statement
	Cause	A CSEG pseudo instruction is described where it cannot be described.
	Action	Move the CSEG pseudo instruction to an appropriate position, or delete it if not necessary.
F173	Message	Impossible to write in table block
	Cause	An instruction which cannot be described in a table block is described in a table block.
	Action	Delete the instruction.
F178	Message	The device doesn't have a system segment
	Cause	An instruction which requires a system segment is used.
	Action	Replace the instruction with another instruction.
F182	Message	Specified illegal symbol name
	Cause	A segment name, section name, or table name is used as a symbol.
	Action	Define another symbol and use that symbol instead of the segment, section, or table name.
F183	Message	Invalid MACRO place
	Cause	• Another macro is defined within the definition of a macro. • A macro is defined within a repetitive pseudo instruction.
	Action	Define the macro where appropriate, other than the above.

Message No.	Description	
F184	Message	No entry characters for LITERAL
	Cause	A string which has not been registered with LITERAL is specified.
	Action	Delete the string if not necessary.
F185	Message	Invalid statement in symbol definition file (.EQU)
	Cause	The symbol definition file contains a statement (instruction) which cannot be described.
	Action	Delete the statement (instruction).
W186 (4)	Message	Nibble number unmatch. Unified the number to left
	Cause	In an extended instruction, the number of nibbles of the first operand differs from that of the second operand.
	Action	Match the number of nibbles.
F187	Message	No mask-option definition file <filename>
	Cause	No mask option definition file exists.
	Action	Prepare a mask option definition file.
F189	Message	Impossible to make SYM file due to error in EQU file
	Cause	No SYM can be created because an error occurred in the EQU file.
	Action	Eliminate all errors in the EQU file.
F190	Message	Impossible to write a local symbol
	Cause	A local symbol is specified for PUBLIC.
	Action	Delete the symbol.
F191	Message	File name error
	Cause	A string other than file names is specified for INCLUDE.
	Action	Specify a valid file name.
F192	Message	Illegal operand type
	Cause	Any of the specified operands has a type which differs from that for the other operands.
	Action	Specify a valid type.
F193	Message	Illegal third operand type
	Cause	The third operand has a type which differs from that for the other operands.
	Action	Specify a valid type.
F194	Message	Illegal fourth operand type
	Cause	The fourth operand has a type which differs from that for the other operands.
	Action	Specify a valid type.

Message No.	Description	
F195	Message	Invalid ENDSUM statement
	Cause	An ENDSUM statement is found although there is no corresponding SUMMARY statement.
	Action	Insert a SUMMARY statement where appropriate. Alternatively, delete the ENDSUM statement if not necessary.
F197	Message	Section or table block multi defined in a source file
	Cause	Two or more section or table blocks having the same name are defined within a source file.
	Action	Combine the blocks into a single section or table.
W198 (1)	Message	No end mark for tag
	Cause	String ;.. corresponding to ;. or ;.V cannot be found.
	Action	Insert ;.. where appropriate.
F199	Message	Invalid SUMMARY statement
	Cause	A SUMMARY statement is described in a section or table block.
	Action	Describe the SUMMARY statement outside a section or table block.
F200	Message	Invalid tag statement
	Cause	• A tag is described outside a section block. • A tag end character (..) is found although there is no corresponding tag start character (.).
	Action	Describe the tag in a section block.
W201 (1)	Message	No mnemonic to make an object code in a tag
	Cause	A tag does not contain an instruction which generates an object code. SIMPLEHOST cannot display data for such a tag.
	Action	When describing a tag, include all instructions which generate an object code, within the tag.

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

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

South America
NEC do Brasil S.A.
Fax: +55-11-6465-6829

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

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

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-250-3583

Japan
NEC Corporation
Semiconductor Solution Engineering Division
Technical Information Support Dept.
Fax: 044-548-7900

I would like to report the following error/make the following suggestion:

Document title: _____

Document number: _____ Page number: _____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	❑	❑	❑	❑
Technical Accuracy	❑	❑	❑	❑
Organization	❑	❑	❑	❑