To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

Notice

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

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

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

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

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

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

User's Manual

CubeSuite Ver. 1.00

Integrated Development Environment

78K0R Coding

Target Device
78K0R Microcontrollers

Document No. U19382EJ1V0UM00 (1st edition)
Date Published September 2008

© NEC Electronics Corporation 2008
Printed in Japan

[MEMO]

SUMMARY OF CONTENTS

CHAPTER 1 GENERAL ... 18

CHAPTER 2 FUNCTIONS ... 30

CHAPTER 3 COMPILER LANGUAGE SPECIFICATIONS ... 55

CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICTION ... 194

CHAPTER 5 LINK DIRECTIVE SPECIFICATION ... 566

CHAPTER 6 FUNCTION SPECIFICATION ... 573

CHAPTER 7 STARTUP ... 760

CHAPTER 8 ROMIZATION ... 775

CHAPTER 9 REFERENCING COMPILER AND ASSEMBLER ... 776

CHAPTER 10 CAUTIONS ... 784

APPENDIX A INDEX ... 792

All trademarks or registered trademarks in this document are the property of their respective owners.

- **The information in this document is current as of September, 2008. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC Electronics does not assume any liability for infringement of patents, copyrights or other intellectual property rights of third parties by or arising from the use of NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".
  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.
  "Standard": Computers, office equipment, communications equipment, test and measurement equipment, audio and visual equipment, home electronic appliances, machine tools, personal electronic equipment and industrial robots.
  "Special": Transportation equipment (automobiles, trains, ships, etc.), traffic control systems, anti-disaster systems, anti-crime systems, safety equipment and medical equipment (not specifically designed for life support).
  "Specific": Aircraft, aerospace equipment, submersible repeaters, nuclear reactor control systems, life support systems and medical equipment for life support, etc.

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

(Note)
(1) "NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.
(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02. 11-1

PREFACE

This manual describes the role of the CubeSuite integrated development environment for developing applications and systems for 78K0R microcontrollers and provides an outline of its features.

CubeSuite is an integrated development environment (IDE) for 78K0R microcontrollers, integrating the necessary tools for the development phase of software (e.g. design, implementation, and debugging) into a single platform.

By providing an integrated environment, it is possible to perform all development using just this product, without the need to use many different tools separately.

Readers

This manual is intended for users who wish to understand the functions of the CubeSuite and design software and hardware application systems.

Purpose

This manual is intended to give users an understanding of the functions of the Cubesuite to use for reference in developing the hardware or software of systems using these devices.

Organization

This manual can be broadly divided into the following units.

CHAPTER 1 GENERAL
CHAPTER 2 FUNCTIONS
CHAPTER 3 COMPILER LANGUAGE SPECIFICATIONS
CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICTION
CHAPTER 5 LINK DIRECTIVE SPECIFICATION
CHAPTER 6 FUNCTION SPECIFICATION
CHAPTER 7 STARTUP
CHAPTER 8 ROMIZATION
CHAPTER 9 REFERENCING COMPILER AND ASSEMBLER
CHAPTER 10 CAUTIONS
APPENDIX A INDEX

How to Read This Manual

It is assumed that the readers of this manual have general knowledge of electricity, logic circuits, and microcontrollers.

Conventions

Data significance:	Higher digits on the left and lower digits on the right
Active low representation:	$\overline{XXX}$ (overscore over pin or signal name)
Note:	Footnote for item marked with Note in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numeric representation:	Decimal … XXXX Hexadecimal … 0xXXXX

Related Documents

The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Document Name		Document No.
CubeSuite Ver.1.00 Integrated Development Environment User's Manual	Start	U19377E
	Programming	U19390E
	Message	U19391E
	78K0R Coding	This document
	78K0R Build	U19385E
	78K0R Debug	U19388E
	78K0R Design	U19379E

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

[MEMO

TABLE OF CONTENTS

CHAPTER 1 GENERAL ... 18

1.1 Outline ... 18
- 1.1.1 C compiler and assembler ... 18
- 1.1.2 Position of compiler and assembler ... 21
- 1.1.3 Processing flow ... 22
- 1.1.4 Basic structure of C source program ... 23

1.2 Features ... 25
- 1.2.1 Features of C compiler ... 25
- 1.2.2 Features of assembler ... 27
- 1.2.3 Limits ... 27

CHAPTER 2 FUNCTIONS ... 30

2.1 Variables (Assembly Language) ... 30
- 2.1.1 Defining variables with no initial values ... 30
- 2.1.2 Defining const constants with initial values ... 30
- 2.1.3 Defining 1-bit variables ... 30
- 2.1.4 1/8 bit access of variable ... 31
- 2.1.5 Allocating to sections accessible with short instructions ... 32

2.2 Variables (C Language) ... 33
- 2.2.1 Allocating data only of reference in ROM ... 33
- 2.2.2 Allocating to sections accessible with short instructions ... 33
- 2.2.3 Allocating in near areas ... 34
- 2.2.4 Allocating in far areas ... 34
- 2.2.5 Allocating addresses directly ... 35
- 2.2.6 Defining 1-bit variables ... 36
- 2.2.7 Empty area of the structure is stuffed ... 36

2.3 Functions ... 37
- 2.3.1 Allocating to sections accessible with short instructions ... 37
- 2.3.2 Allocating in near areas ... 37
- 2.3.3 Allocating in far areas ... 38
- 2.3.4 Allocating addresses directly ... 38
- 2.3.5 Inline expansion of function ... 39
- 2.3.6 Embedding assembly instructions ... 39

2.4 Using Microcontroller Functions ... 40
- 2.4.1 Accessing special function registers (SFR) from C ... 40
- 2.4.2 Interrupt functions in C ... 41
- 2.4.3 Using CPU control instructions in C ... 42

2.5 Startup Routine ... 44
- 2.5.1 Deleting unused functions and areas from startup routine ... 44
- 2.5.2 Allocating stack area ... 45
- 2.5.3 Initializing RAM ... 45

2.6 Link Directives ... 46
- 2.6.1 Partitioning default areas ... 46
- 2.6.2 Specifying section allocation ... 46

2.7 Reducing Code Size ... 47

2.7.1 Using extended functions to generate efficient object code ... 47
2.7.2 Calculating complex expressions ... 50
2.8 Compiler and Assembler Mutual References ... 51
2.8.1 Mutually referencing variables ... 51
2.8.2 Mutually referencing functions ... 53

CHAPTER 3 COMPILER LANGUAGE SPECIFICATIONS ... 55

3.1 Basic Language Specifications ... 55
3.1.1 Processing system dependent items ... 55
3.2 Environment During Compilation ... 66
3.2.1 Internal representation and value area of data ... 66
3.2.2 Memory ... 70
3.3 Extended Language Specifications ... 72
3.3.1 Macro names ... 72
3.3.2 Keywords ... 72
3.3.3 #pragma directives ... 74
3.3.4 Using extended functions ... 75
3.3.5 C source modifications ... 177
3.4 Function Call Interface ... 178
3.4.1 Return values ... 178
3.4.2 Ordinary function call interface ... 178
3.5 List of saddr Area Labels ... 182
3.6 List of Segment Names ... 184
3.6.1 List of segment names ... 184
3.6.2 Segment allocation ... 186
3.6.3 C source example ... 186
3.6.4 Example of output assembler module ... 186

CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICTION ... 194

4.1 Description Methods of Source Program ... 194
4.1.1 Basic configuration ... 194
4.1.2 Description method ... 200
4.1.3 Expressions and operators ... 210
4.1.4 Arithmetic operators ... 213
4.1.5 Logic operators ... 221
4.1.6 Relational operators ... 226
4.1.7 Shift operators ... 233
4.1.8 Byte separation operators ... 236
4.1.9 Word separation operators ... 239
4.1.10 Special operators ... 242
4.1.11 Other operator ... 246
4.1.12 Restrictions on operations ... 248
4.1.13 Absolute expression definitions ... 252
4.1.14 Bit position specifier ... 252
4.1.15 Identifiers ... 254
4.1.16 Operand characteristics ... 254
4.2 Directives ... 262
4.2.1 Overview ... 262
4.2.2 Segment definition directives ... 263
4.2.3 Symbol definition directives ... 279

4.2.4 Memory initialization, area reservation directives ... 286
4.2.5 Linkage directives ... 296
4.2.6 Object module name declaration directive ... 303
4.2.7 Branch instruction automatic selection directives ... 305
4.2.8 Macro directives ... 310
4.2.9 Assemble termination directive ... 325
4.3 Control Instructions ... 327
4.3.1 Overview ... 327
4.3.2 Assemble target type specification control instruction ... 328
4.3.3 Debug information output control instructions ... 330
4.3.4 Cross-reference list output specification control instructions ... 335
4.3.5 Include control instruction ... 340
4.3.6 Assembly list control instructions ... 344
4.3.7 Conditional assembly control instructions ... 366
4.3.8 Kanji code control instruction ... 389
4.3.9 Other control instructions ... 391
4.4 Macros ... 392
4.4.1 Overview ... 392
4.4.2 Using macros ... 392
4.4.3 Symbols in macros ... 395
4.4.4 Macro operators ... 397
4.5 Reserved Words ... 398
4.6 Instructions ... 399
4.6.1 Differences from 78K0 microcontrollers (for assembler users) ... 399
4.6.2 Memory space ... 400
4.6.3 Registers ... 403
4.6.4 Addressing ... 408
4.6.5 Instruction set ... 418
4.6.6 Explanation of instructions ... 449
4.6.7 Pipeline ... 563

CHAPTER 5 LINK DIRECTIVE SPECIFICATION ... 566

5.1 Coding Method ... 566
5.1.1 Link directives ... 566
5.2 Reserved Words ... 570
5.3 Coding Examples ... 571
5.3.1 When specifying link directive ... 571
5.3.2 When using compiler ... 572

CHAPTER 6 FUNCTION SPECIFICATION ... 573

6.1 Distribution Libraries ... 573
6.1.1 Standard library ... 574
6.1.2 Runtime library ... 579
6.2 Interface Between Functions ... 584
6.2.1 Arguments ... 584
6.2.2 Return values ... 584
6.2.3 Saving registers used by separate libraries ... 584
6.3 Header Files ... 586
6.3.1 ctype.h ... 586
6.3.2 setjmp.h ... 586

6.3.3 stdarg.h ... 586
6.3.4 stdio.h ... 586
6.3.5 stdlib.h ... 587
6.3.6 string.h ... 587
6.3.7 error.h ... 587
6.3.8 errno.h ... 588
6.3.9 limits.h ... 588
6.3.10 stddef.h ... 589
6.3.11 math.h ... 589
6.3.12 float.h ... 590
6.3.13 assert.h ... 592
6.4 Re-entrant ... 592
6.5 Use of Arguments/Return Values Suitable for Standard Library ... 593
6.6 Character/String Functions ... 594
6.7 Program Control Functions ... 614
6.8 Special Functions ... 617
6.9 Input and Output Functions ... 622
6.10 Utility Functions ... 640
6.11 String and Memory Functions ... 672
6.12 Mathematical Functions ... 695
6.13 Diagnostic Function ... 742
6.14 Library Stack Consumption List ... 744
6.14.1 Standard libraries ... 744
6.14.2 Runtime libraries ... 749
6.15 List of Maximum Interrupt Disabled Times for Libraries ... 755
6.16 Batch Files for Update of Startup Routine and Library Functions ... 756
6.16.1 Using batch files ... 757

CHAPTER 7 STARTUP ... 760

7.1 Function Overview ... 760
7.2 File Organization ... 760
7.2.1 "bat" folder contents ... 761
7.2.2 "lib" folder contents ... 761
7.2.3 "src" folder contents ... 763
7.3 Batch File Description ... 763
7.3.1 Batch files for creating startup routines ... 763
7.4 Startup Routines ... 764
7.4.1 Overview of startup routines ... 764
7.4.2 Startup routine preprocessing ... 766
7.4.3 Startup routine initial settings ... 768
7.4.4 Startup routine main function startup and postprocessing ... 771
7.5 ROMization Processing in Startup Module for Flash Area ... 772
7.6 Coding Examples ... 773
7.6.1 When revising startup routine ... 773
7.6.2 When using RTOS ... 774

CHAPTER 8 ROMIZATION ... 775

CHAPTER 9 REFERENCING COMPILER AND ASSEMBLER ... 776

9.1 Accessing Arguments and Automatic Variables ... 776
9.2 Register HL is used as a base pointer for accessing arguments and automatic variables stored in a stack.Storing Return Values ... 776
9.3 Calling Assembly Language Routines from C Language ... 776
9.3.1 C language function calling procedure ... 776
9.3.2 Saving data from assembly language routine and returning ... 777
9.4 Calling C Language Routines from Assembly Language ... 779
9.4.1 Calling C language function from assembly language program ... 779
9.5 Referencing Variables Defined in C Language ... 781
9.6 Referencing Variables Defintед in Assembly Language from C Language ... 782
9.7 Points of Caution for Calling Between C Language Functions and Assembler Functions ... 782

CHAPTER 10 CAUTIONS ... 784

APPENDIX A INDEX ... 792

LIST OF FIGURES

Figure No. Title, Page

1-1 Flow of Development with C Compiler and Assembler ... 18

1-2 Division into Modules ... 19

1-3 Resource Utilization ... 20

1-4 Flow of microcontroller Application Product Development ... 21

1-5 Flow of Compiler and Assembler Processing ... 22

2-1 Stack Setting ... 45

3-1 Floating Point Number Format ... 67

3-2 Usage of Memory Space ... 71

3-3 Memory Map Example 1 ... 149

3-4 Memory Map Example 2 ... 149

3-5 Memory Map Example 3 ... 150

3-6 Memory Map Examplee 4 ... 150

3-7 Memory Map Example 5 ... 150

3-8 Memory Map Example 6 ... 151

4-1 Source Module Configuration ... 194

4-2 Overall Configuration of Source Program ... 196

4-3 Examples of Source Module Configurations ... 196

4-4 Sample Source Program Configuration ... 197

4-5 Statement Fields ... 200

4-6 Segment Memory Mapping ... 264

4-7 Relationship of Symbols Between 2 Modules ... 296

4-8 Memory Maps of 78K0 and 78K0R Microcontrollers ... 400

4-9 Mirror Area Examples ... 401

4-10 Configuration of Program Counter ... 403

4-11 Configuration of Program Status Word ... 403

4-12 Configuration of Stack Pointer ... 404

4-13 Data Saved to Stack Memory ... 405

4-14 Configurations of ES and CS Registers ... 407

4-15 Configuration of Processor Mode Control Register ... 407

4-16 How Relative Addressing Works ... 408

4-17 Example of CALL !!addr20/BR !!addr20 Addressing ... 408

4-18 Example of CALL !addr16/BR !addr16 Addressing ... 408

4-19 How Table Indirect Addressing Works ... 409

4-20 How register Direct Addressing Works ... 409

4-21 How Implied Addressing Works ... 410

4-22 How Register Addressing Works ... 410

4-23 Example of ADDR16 Addressing ... 411

4-24 Example of ES:ADDR16 Addressing ... 411

4-25 How Short Direct Addressing Works ... 412

4-26 How SFR Addressing Works ... 412

4-27 Example of [DE] and [HL] Addressing ... 413
4-28 Example of ES:[DE] and ES:[HL] Addressing ... 413
4-29 Example of [SP+byte] Addressing ... 414
4-30 Example of [HL+byte] and [DE+byte] Addressing ... 414
4-31 Example of word[B] and word[C] Addressing ... 415
4-32 Example of word[BC] Addressing ... 415
4-33 Example of ES:[HL+byte] and ES:[DE+byte] Addressing ... 415
4-34 Example of ES:word[B] and ES:word[C] Addressing ... 416
4-35 Example of ES:word[BC] Addressing ... 416
4-36 Example of [HL+B] and [HL+C] Addressing ... 417
4-37 Example of ES:[HL+B] and ES:[HL+C] Addressing ... 417
4-38 Pipeline Execution of Five Typical Instructions (Example) ... 563
5-1 Example of Memory Area Name that Cannot Be Specified ... 568
5-2 Specified Start Address and Segment Location ... 569
5-3 Example Allocations of Input Segment SEG1 ... 571
6-1 Stack Area During Function Call ... 585
6-2 Syntax of format Commands ... 626
6-3 Syntax of format Commands ... 630
7-1 Programs Related to Startup Routines and Their Configurations ... 765
7-2 ROMization Processing for External Variable with Initial Value ... 770
7-3 Startup Routine Initial Settings Example ... 774
9-1 Stack Immediately After Function Is Called ... 777
9-2 Setting Return Value ... 779
9-3 Returning to Main Function ... 779
9-4 Argument Passing ... 780
9-5 Referencing Return Values ... 781
9-6 Argument Positions on Stack ... 783

LIST OF TABLES

Table No. Title, Page

1-1 Programs Involved in Processing Flow ... 22

1-2 Methods to Improve Execution Speed ... 25

1-3 Assembler Translation Limits ... 27

1-4 Linker Limits ... 29

3-1 Data types and Sizes ... 55

3-2 Expanded Notation and Meaning ... 56

3-3 Translation Limit Values ... 57

3-4 Limit Values of General Integer Type (limits.h File) ... 58

3-5 Definition of Limit Values of Floating-point type (float.h File) ... 59

3-6 List of Supported Macros ... 65

3-7 Definition of NULL, size_t, ptrdiff_t (stddef.h File) ... 65

3-8 Value Ranges by Type ... 67

3-9 Numerical Exception ... 69

3-10 Memory Models ... 70

3-11 Keywords Added by 78K0R C Compiler ... 72

3-12 Extended Function List ... 75

3-13 Handling of ROM Data When There Is No Mirror of Boot Area ... 148

3-14 Handling of ROM Data When There Is No Mirror of Flash Area ... 149

3-15 Handing of Functions When Start of Flash Area Is Not Within 64 KB ... 149

3-16 Storage Locations of Return Values ... 178

3-17 Location Where First Argument Is Passed (Function Calling Side) ... 178

4-1 Instructions That Can Appear in Module Headers ... 195

4-2 Alphanumeric Characters ... 201

4-3 Special Characters ... 201

4-4 Operator Types ... 210

4-5 Operator Precedence Levels ... 211

4-6 Relocation Attribute Types ... 248

4-7 Combinations of Operators and Terms by Relocation Attribute (Relocatable Terms) ... 248

4-8 Combinations of Operators and Terms by Relocation Attribute (External Reference Terms) ... 250

4-9 Symbol Attribute Types in Operations ... 250

4-10 Combinations of Operators and Terms by Symbol Attribute ... 251

4-11 Instruction Operand Value Ranges ... 254

4-12 Value ranges of Directive Operands ... 259

4-13 Properties of Described Symbols as Operands ... 260

4-14 Properties of Described Symbols as Operands of Directives ... 261

4-15 List of Directives ... 262

4-16 Segment Definition Method and Memory Address Location ... 263

4-17 Relocation Attributes of CSEG ... 266

4-18 Relocation Attributes of DSEG ... 270

4-19 Relocation Attributes of BSEG ... 274

4-20 Control Instruction List ... 327
4-21 Control Instructions and Assembler Options ... 327
4-22 Fields Where Reserved Words Can Appear ... 398
4-23 List of Reserved Words ... 398
4-24 Stack Data Size Differences Between 78K0 and 78K0R Microcontrollers ... 404
4-25 List of General-purpose Registers (78K0 Compatible) ... 406
4-26 List of Fixed-address SFRs ... 407
4-27 Operand Type Representations and Source Code Formats ... 418
4-28 Operation Field Symbols ... 419
4-29 Flag Field Symbols ... 420
4-30 Examples of PREFIX Instructions in Use ... 420
4-31 Operation List (8-bit Data Transfer Instructions) ... 421
4-32 Operation List (16-bit Data Transfer Instructions) ... 425
4-33 Operation List (8-bit Manipulation Instructions) ... 427
4-34 Operation List (16-bit Operation Instructions) ... 432
4-35 Operation List (Multiply Instructions) ... 434
4-36 Operation List (Increase-Decrease Instructions) ... 435
4-37 Operation List (Shift Instructions) ... 437
4-38 Operation List (Rotate Instructions) ... 438
4-39 Operation List (Bit Manipulation Instructions) ... 439
4-40 Operations List (Call Return Instructions) ... 441
4-41 Operation List (Stack Manipulation Instructions) ... 443
4-42 Operation List (Unconditional Branch Instructions) ... 444
4-43 Operation List (Conditional Branch Instructions) ... 445
4-44 Operation List (Conditional Step Instruction) ... 447
4-45 Operation List (CPU Control Instruction) ... 448
4-46 Assembly Language Instruction List ... 449
6-1 Distribution Libraries ... 573
6-2 Maximum Interrupt Disabled Time (Number of Clocks) for Libraries ... 755
6-3 Batch Files for Updating Library Functions ... 756
7-1 "bat" Folder Contents ... 761
7-2 "lib" folder Contents ... 761
7-3 "src" Folder Contents ... 763
7-4 ROM Area Section for Initialization Data ... 772
7-5 RAM Area Section of Copy Destination ... 773

CHAPTER 1 GENERAL

This chapter explains the roles of the 78K0R C compiler package (called "CA78K0R") in system development, and provides an outline of their functions.

1.1 Outline

78K0R C compiler is a translation program that converts source programs written in traditional C or ANSI C into machine language. 78K0R C compiler can produce either object files or assembly source files.

78K0R assembler is a language processing program that converts source programs written in assembly language into machine language.

1.1.1 C compiler and assembler

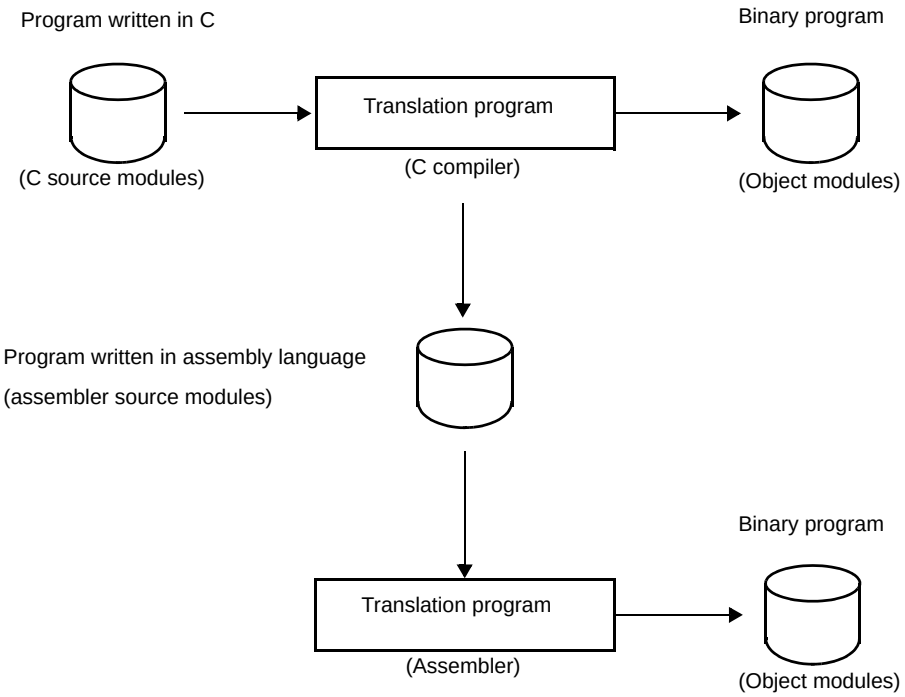
(1) C language and assembly language

A C compiler takes C source modules as input and produces either object modules or assembly source modules as output. This means that you can develop your programs in C and use assembly language as required to make fine adjustments.

An assembler takes assembly source modules as input and produces object modules as output.

The following figure shows the flow of program development with a C compiler and an assembler.

Figure 1-1. Flow of Development with C Compiler and Assembler



(2) Relocatable assemblers

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

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

Depending on the method used to allocate machine language instructions to memory addresses, assemblers can be broadly divided into absolute assemblers and relocatable assemblers. RA78K0R is a relocatable assembler.

- Absolute assembler

  An absolute assembler allocates machine language instructions assembled from the assembly language at absolute addresses.

- Relocatable assembler

  In a relocatable assembler, the addresses determined for the machine language instructions assembled from the assembly language are tentative

  Absolute addresses are determined subsequently by the linker.

In the past, when programs were created with absolute assemblers, programmers normally had to write the entire program as a single large block. However, when all the components of a large program are contained in a single block, the program becomes complicated, making it harder to understand and maintain.

To avoid this, large programs are now usually developed by dividing them into several subprograms, called modules, one for each functional unit. This programming technique is called modular programming.

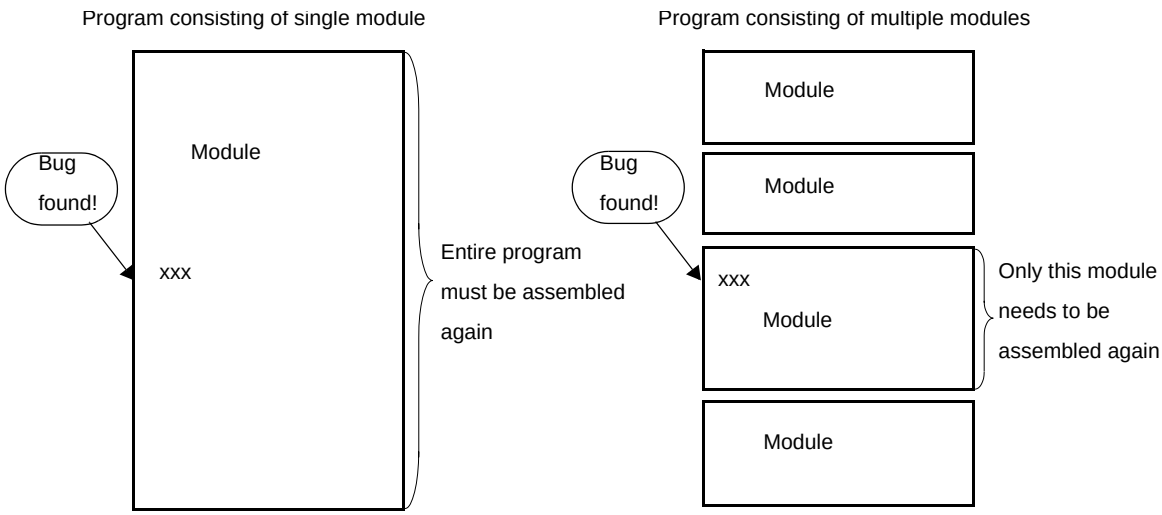
Relocatable assemblers are well suited for modular programming, which has the following advantages:

(a) Greater development efficiency

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

Moreover, when bugs are found, only the module that contained the bugs needs to be corrected and assembled again, instead of needing to assemble the entire program. This shortens debugging time.

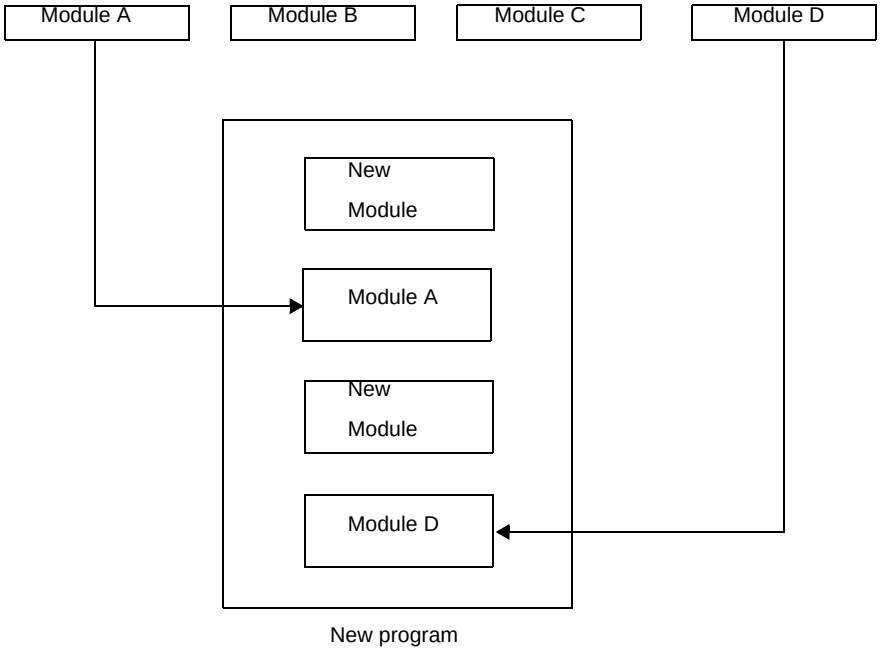
Figure 1-2. Division into Modules



(b) Utilization of resources

Reliable and versatile modules from previous development efforts are software resources that can be reused in new programs. As you accumulate more of these resources, you save time and labor in developing new programs.

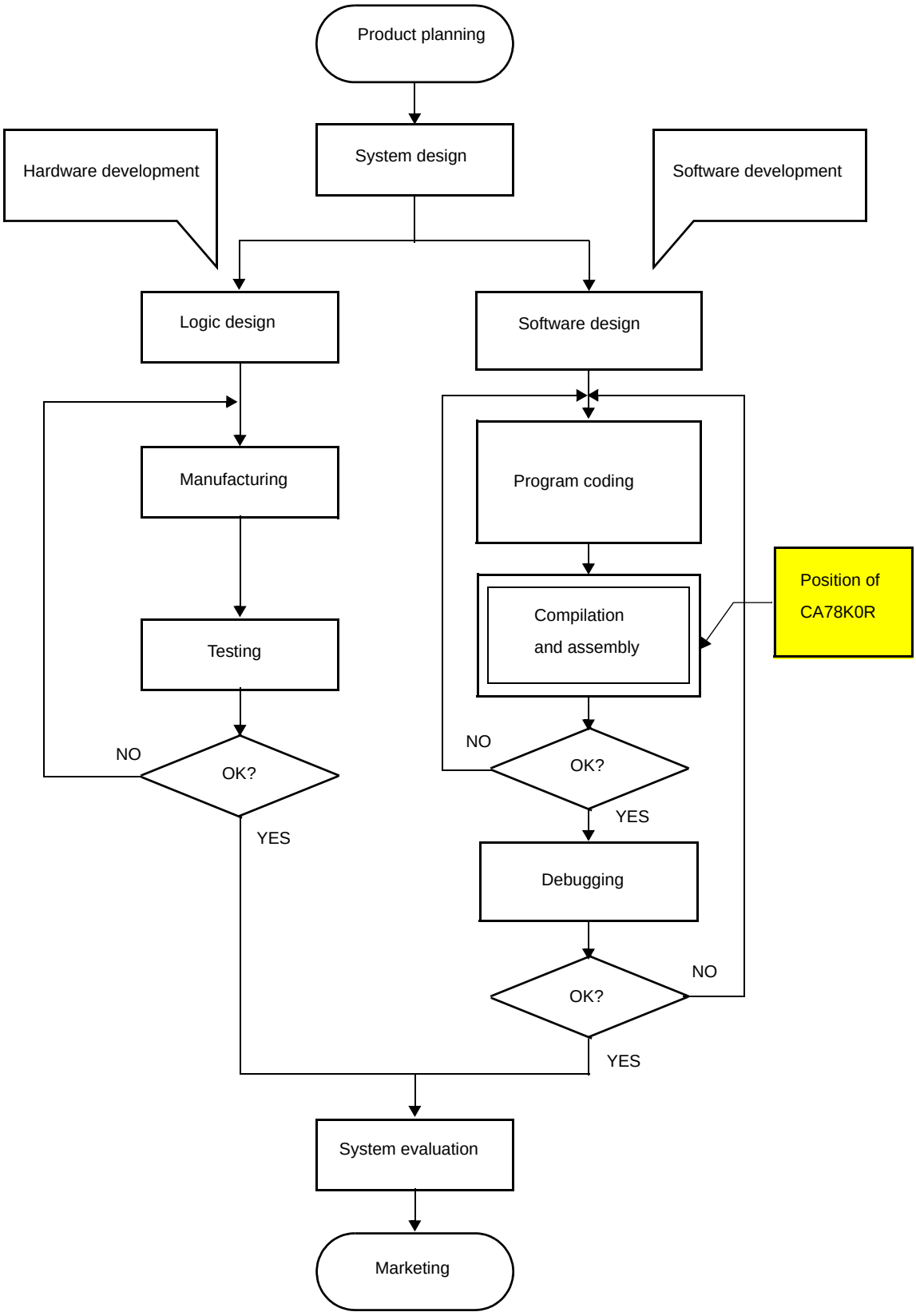
Figure 1-3. Resource Utilization



1.1.2 Position of compiler and assembler

The following figure shows the position of compiler and assembler in the flow of product development.

Figure 1-4. Flow of microcontroller Application Product Development



1.1.3 Processing flow

This section explains the flow of processing in program development.

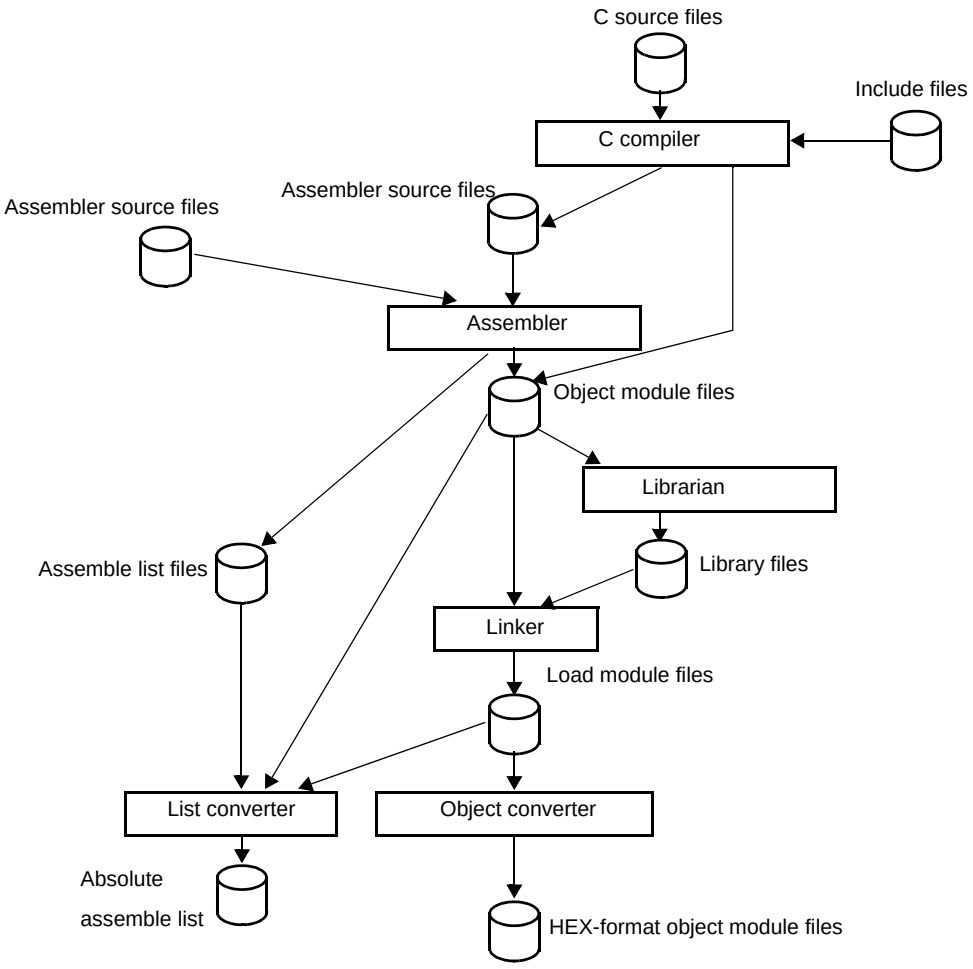
The C compiler compiles C source module files and generates object files or assembly source module files. By hand optimizing the generated assembly source module files, you can create more efficient object module files. This is useful when the program must perform high-speed processing and when compact modules are desirable.

The following programs are involved in the processing flow.

Table 1-1. Programs Involved in Processing Flow

Program	Function
Compiler	Compiles C source module files
Assemble	Assembles assembly language source module files
Linker	Links object module files Determines addresses of relocatable segments
Object converter	Converts to HEX-format object module files
Librarian	Creates library files
List converter	Generates absolute assemble list files

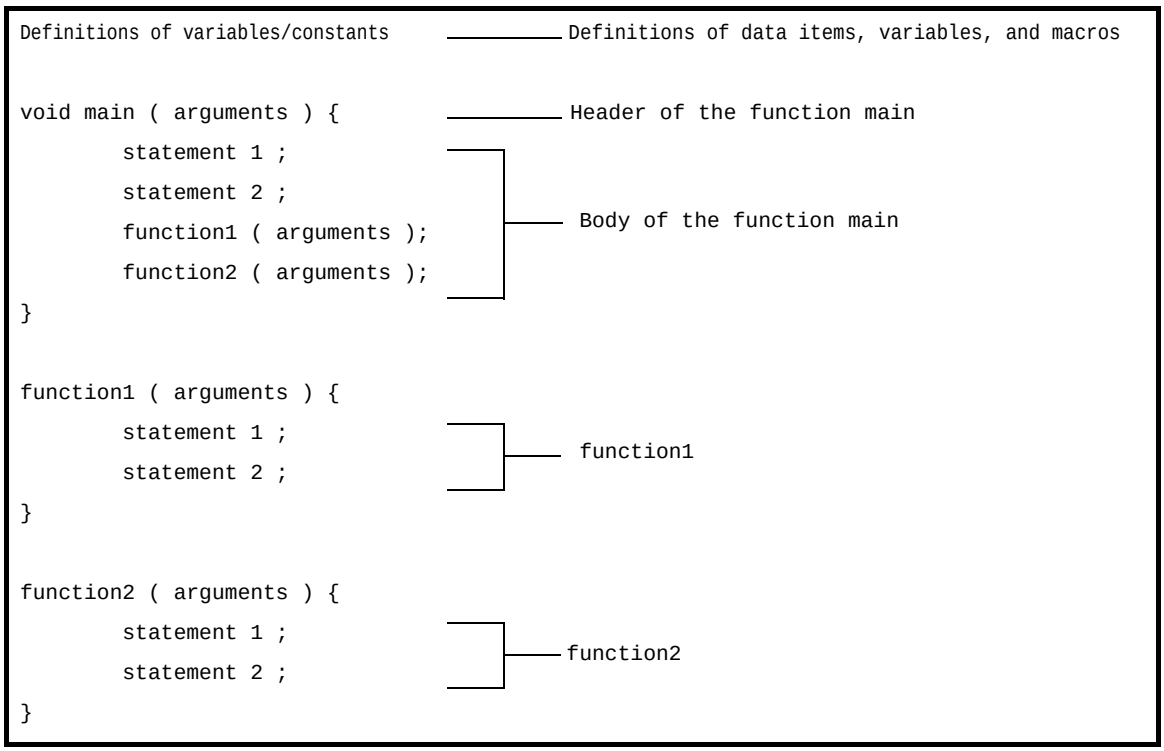
Figure 1-5. Flow of Compiler and Assembler Processing



1.1.4 Basic structure of C source program

A program in C is a collection of functions. A function is an independent unit of code that performs a specific action. Every C language program must have a function "main" which becomes the main routine of the program and is the first function to be called when execution begins.

Each function consists of a header part, which defines its function name and arguments, and a body part, which consists of declarations and statements. The format of C programs is shown below.



An actual C source program looks like this.

```
#define TRUE    1       /* #define xxx xxx  Preprocessor directive (macro definition) */
#define FALSE   0       /* #define xxx xxx  Preprocessor directive (macro definition) */
#define SIZE    200     /* #define xxx xxx  Preprocessor directive (macro definition) */

void displaystring ( char*, int ); /* xxx xxxx ( xxx, xxx )
                                      Function prototype declaration */
void displaychar ( char );         /* xxx xxxx ( xxx )
                                      Function prototype declaration */

char    mark[SIZE + 1];              /* char xxx
                                      Type declaration, External definition */
                                     /* xx[xx]           Operator */
void main ( void ) {
    int     i, prime, k, count ;     /* int xxx          Type declaration */

    count = 0 ;                      /* xx = xx          Operator */

    for ( i = 0 ; i <= SIZE ; i ++ )   /* for ( xx ; xx ; xx ) xxx ; Control structure */
        mark[i] = TRUE ;
    for ( i = 0 ; i <= SIZE ; i ++ ) {
```

```
        if ( mark[i] ) {
            prime = i + i + 3 ;             /* xxx = xxx + xxx + xxx   Operator */
            displaystring ( "%6d", prime );  /* xxx ( xxx );           Operator */

            count ++ ;
            if ( ( count%8 ) == 0 ) displaychar ( '\n' );  /* if ( xxx ) xxx ;
                                                                Control structure */
            for ( k = i + prime ; k <= SIZE ; k += prime )
                mark[k] = FALSE ;
        }
    }
    displaystring ( "\n%d primes found.", count );       /* xxx ( xxx );      Operator */
}

void displaystring ( char *s, int i ) {
    int     j ;
    char    *ss ;

    j = i ;
    ss = s ;
}

void displaychar ( char c ) {
    char    d ;

    d = c ;
}
```

(1) Declaration of type and storage class

Declares the data type and storage class of an object identifier.

(2) Operator or expression

Performs arithmetic, logical, or assignment operations.

(3) Control structure

Specifies the flow of the program. C has a number of instructions for different types of control, such as conditional control, iteration, and branching.

(4) Structure or union

Declares a structure or union. A structure is a data object that contains several subobjects or members that may have different types. A union is like a structure, but allows two or more variables to share the same memory.

(5) External definition

Declares a function or external object. A function is an independent unit of code that performs a specific action. A C program is a collection of functions.

(6) Preprocessor directive

An instruction to the compiler. The #define directive instructs the compiler to replace any instances of the first operand that appear in the program with the second operand.

(7) Declaration of function prototype

Declares the types of the return value and arguments of a function.

1.2 Features

This section explains the features of the CA.

1.2.1 Features of C compiler

(1) Conforms to ANSI C

The compiler conforms to the ANSI standard for the C language.

Remark ANSI: American National Standards Institute

(2) Designed for efficient use of ROM and RAM memory

- External variables can be allocated to short direct addressing memory.
  Function arguments and auto variables can be allocated to short direct addressing memory or registers.
- Bit instructions enable definitions and operations on data in units of 1 bit.

(3) Interrupt control features

- Peripheral hardware of 78K0R can be controlled directly from C.
- Interrupt handlers can be written directly in C.

(4) Supports extended functions of 78K0R

78K0R C compiler supports the following extended functions, which are not defined in the ANSI standard. Some of these functions allow special-purpose registers to be accessed in C, while others enable more compact object code and higher execution speed.

The following table lists extended functions that reduce the size of object code and improve execution speed.

Table 1-2. Methods to Improve Execution Speed

Method	Extended Function
Allocate variables to registers	Register variables
Allocate variables to the saddr area	sreg/__sreg
Use sfr names.	sfr area
Embed assembly language statements in C source programs.	ASM statements
Accessing the saddr or sfr area can be made on a bit-by-bit basis.	bit type variables, boolean/__boolean type variables
Specify bit fields using the unsigned char type.	Bit field declarations
The code to multiply can be directly output with inline expansion.	Multiplication function
The code to rotate can be directly output with inline expansion.	Rotation functions
Specific data and instructions can be directly embedded in the code area.	Data insertion function
memcpy and memset are directly expanded inline and output.	Memory function

See "3.3 Extended Language Specifications" for detailed information about the extended functions of the 78K0R C compiler.

1.2.2 Features of assembler

The 78K0R assembler has the following features.

(1) Macro function

When the same group of instructions occurs in a source program over and over again, you can define a macro to give a single name to the group of instructions.
Macros can increase your coding efficiency and make your programs more readable.

(2) Optimized branching directives

The 78K0R assembler provides the BR and CALL (Branch instruction automatic selection directives).
A characteristic of programs that make efficient use of memory is selection of the appropriate branching instructions, using only the number of bytes required by the branch destination range. But it is a burden for the programmer to need to be conscious of the branch destination range for every branch. The BR and CALL directives are automatic solutions to this problem. They facilitate memory-efficient branching by instructing the assembler to generate the most appropriate branching instruction for the branch destination range. This function is called branch instruction optimization.

(3) Conditional assembly

Conditional assembly allows you to specify conditions that determine whether or not specific sections of the source program are assembled.
For example, when the source contains debugging statements, a switch can be set to determine whether or not they should be translated into machine language. When they are no longer needed, they can be excluded from the output with no major modifications to the source program.

(4) 78K0 compatibility macro function

The assembly of the assembler source file made by the assembler for 78K0 is enabled.
When assemble 78K0 instructions that cannot be used on 78K0R without changing the description of the source, specify the -compati option.

78K0 instructions that cannot be used on 78K0R: DIVUW/ROR4/ROL4/ADJBA/ADJBS/CALLF/DBNZ

1.2.3 Limits

(1) Compiler limits

See "(9) Translation Limit" for the limits of the compiler.

(2) Assembler limits

Table 1-3. Assembler Translation Limits

Description	Limit
Number of symbols (local + public)	65,535
Number of symbols for which cross-reference list can be output	65,534 Note 1
Maximum size of macro body for one macro reference	1 Mbyte
Total size of all macro bodies	10 Mbyte
Number of segments in one file	256
Number of macro and include specifications in one file	10,000
Number of macro and include specifications in one include file	10,000
Number of relocation data items Note 2	65,535
Line number data items	65,535

Description	Limit
Number of BR/CALL directives in one file	32,767
Character length of source line	2,048 Note 3
Character length of symbol	256
Character length of name definition Note 4	1,000
Character length of switch name Note 4	31
Character length of segment name	8
Character length of module name (NAME directive)	256
Number of parameters in MACRO directive	16
Number of arguments in macro reference	16
Number of arguments in IRP directive	16
Number of local symbols in macro body	64
Total number of local symbols in expanded macro	65,535
Nesting levels in macro (macro reference, REPT directive, IRP directive)	8 levels
Number of characters in TITLE control instruction (-lh option)	60 Note 5
Number of characters in SUBTITLE control instruction	72
Include file nesting levels in 1 file	16 levels
Conditional assembly nesting levels	8 levels
Number of include file paths specifiable by -i option	64
Number of symbols definable by -d option	30

Notes 1. Excluding the number of module names and section names.
Available memory is used. When memory is insufficient, a file is used.

2. Information passed to the linker when a symbol value cannot be resolved by the assembler.
For example, when an externally referenced symbol is referenced by the MOV instruction, two items of relocation information are generated in a .rel file.

3. Including CR and LF codes. If a line is longer than 2048 characters, a warning message is output and the 2049th and following characters are ignored.

4. Switch names are set to true/false by the SET and RESET directives and are used by constructs such as $If.

5. If the maximum number of characters that can be specified in one line of the assemble list file ("X") is 119, this figure will be "X - 60" or less.

(3) Linker limits

Table 1-4. Linker Limits

Description	Limit
Number of symbols (local + public)	65,535
Line number data items in 1 segment	65,535
Number of segments	65,535[Note]
Number of input modules	1,024
Character length of memory area name	256
Number of memory areas	100[Note]
Number of library files specifiable by the -b option	64
Number of include file paths specifiable by the -i option	64

Note Including those defined by default.

CHAPTER 2 FUNCTIONS

This section explains programming technique to use CA78K0R more effectively and use of extended functions.

2.1 Variables (Assembly Language)

This section explains techniques for using variables in assembly language.

2.1.1 Defining variables with no initial values

Allocate memory area in a data segment.

Use the DSEG quasi directive to define a data segment, and use the DS quasi directive to allocate memory area.

Example Define an 10-byte variable with no initial values.

```
        DSEG
_table: DS      10
```

Remark See "DSEG" and "DS".

2.1.2 Defining const constants with initial values

Initialize memory area in a code segment.

Use the CSEG quasi directive to define a code segment, and use the DB (1 byte), DW (2 bytes), or DG (4 bytes) quasi directive to initialize memory area.

Example Defining constants with initial values

```
        CSEG
_val1:  DB      0F0H    ; 1 byte
_val2:  DW      1234H   ; 2 bytes
_val3:  DG      56789H  ; 4 bytes (20 bits)
```

Remark See "CSEG", "DB", "DW", and "DG".

2.1.3 Defining 1-bit variables

Allocate 1 bit memory area in a bit segment.

Use the BSEG quasi directive to define a bit segment, and use the DBIT quasi directive to allocate 1 bit memory area.

Example Define bit variables with no initial values.

```
        BSEG
_bit1   DBIT
_bit2   DBIT
_bit3   DBIT
```

Remark See "BSEG" and "DBIT".

2.1.4 1/8 bit access of variable

In assembly language source code, give two symbols for the address in the saddr area. To use the symbol name respectively for the bit access and for byte access,specify saddr as the relocation attribute of a DSEG segment, define bit name of a symbol for byte access as a symbol name for bit access by a EQU quasi directives.

Example Byte access symbol name: FLAGBYTE
Bit access symbol name: FLAGBIT

- smp1.asm

```
        NAME    SMP1
        PUBLIC  FLAGBYTE, FLAGBIT

FLAGS   DSEG    SADDR           ; The relocation attribute of DSEG is SADDR
FLAGBYTE: DS (1)                ; Define FLAGBYTE
FLAGBIT EQU     FLAGBYTE.0      ; Define FLAGBIT
        END
```

- smp2.asm

```
        NAME    SMP2

        EXTRN   FLAGBYTE
        EXTBIT  FLAGBIT         ; FLAGBIT declared as EXTBIT

        CSEG
C1:
        MOV     FLAGBYTE, #0FFH
        CLR1    FLAGBIT
        END
```

Remark See "DSEG" and "EQU".

2.1.5 Allocating to sections accessible with short instructions

Compared to other data memory areas, the short direct addressing area can be accessed with shorter instructions. Improve the memory efficiency of programs by efficiently using this area.

To allocate in the short direct addressing area, specify saddr or saddrp as the relocation attribute of a DSEG quasi directive.

The following examples explains use in assembly source code.

- Module 1

```
PUBLIC  TMP1, TMP2
WORK    DSEG saddrp
TMP1:   DS 2 ; word
TMP2:   DS 1 ; byte
```

- Module 2

```
EXTRN   TMP1, TMP2
SAB     CSEG
MOVW    TMP1, #1234H
MOV     TMP2, #56H
        :
```

Remark See "DSEG".

2.2 Variables (C Language)

This section explains Variables (C language).

2.2.1 Allocating data only of reference in ROM

(1) Allocating variables with initial values in ROM

Specify the const qualifier to allocate variables with initial values only of a reference in ROM.

Example Allocating variable "a" with initial values only of a reference in ROM

```
const int a = 0x12 ;  /* Allocating ROM */
int b = 0x12 ;        /* Allocating ROM/RAM */
```

Variable "a" is allocated in ROM.
For variable "b", the initializing value is allocated in ROM and the variable itself is allocated in RAM (areas is required in both ROM and RAM).
Startup routine ROMization, an initial value of ROM is copied in a variable of RAM.
ROMization requires areas in both ROM and RAM.

(2) Allocating table data in ROM

If allocating table data in ROM only, define type qualifier const, as follows.

```
const unsigned char table_data[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

2.2.2 Allocating to sections accessible with short instructions

Compared to other data memory areas, the short direct addressing area can be accessed with shorter instructions. Improve the memory efficiency of programs by efficiently using this area.

The use example is shown below.

External variables defined sreg or __sreg, and static variables within functions (called sreg variables) are automatically allocated in relocatable in short direct addressing area [FFE20H to FFEB3H].

```
extern sreg int hsmm0 ;
extern sreg int hsmm1 ;
extern sreg int *hsptr ;

void main ( void ) {
        hsmm0 -= hsmm1 ;
}
```

Remark See "How to use the saddr area (sreg/__sreg)".

2.2.3 Allocating in near areas

Using the small model, the compiler generates code with 16-bit address lengths.

When knowing in advance that code and data are into 64 KB, obtain more compact code by using the small model instead of the large model.

Specify the small model (-ms option) with the compiler option. Data and functions are allocated in near areas.

Or add the __near type qualifier to variable and function declarations.

```
__near int func ( void );            /* Allocating in near area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__near int func ( void ) {           /* Allocating in near area */
        /* Function processing */
        return 0 ;
}
```

Remark Refer to "near/far area specification".

2.2.4 Allocating in far areas

Using the large model, the compiler generates code with 20-bit address lengths.

If data are into 64 KB and code are into 1MB, use the medium model.

Specify the medium model (-mm option) with the compiler option. Data are allocated in near areas and functions are allocated in far areas.

Or add the __near and __far type qualifier to variable and function declarations.

```
__far  int func ( void );            /* Allocating in far area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__far int func ( void ) {            /* Allocating in far area */
        /* Function processing */
        return 0 ;
}
```

If code and data are into 1 MB, use the large model.

Specify the large model (-ml option) with the compiler option. Data and functions are allocated in far areas.

Or add the __far type qualifier to variable and function declarations.

```
__far int func ( void );             /* Allocating in far area */
__far const int a = 0 x 12 ;         /* Allocating in far area */
__far int b = 0 x 12 ;               /* Allocating in far area */

__far int func ( void ) {            /* Allocating in far area */
        /* Function processing*/
        return 0 ;
}
```

Remark Refer to "near/far area specification".

2.2.5 Allocating addresses directly

(1) direcrtmap

External variable declared __directmap and the initializing value of static variable in functions are allocation address, the variable is mapped to the specified address. Specify the allocation address as an integral number. __directmap variables in C source files are handled as well as static variables.

Make the __directmap declaration in the module which defines the variable that to map to an absolute address.

```
__directmap  char c = 0xffe00 ;
__directmap  __sreg char d = 0xffe20 ;
__directmap  __sreg char e = 0xffe21 ;

__directmap  struct x {
        char a ;
        char b ;
} xx = { 0xffe30 };

void main ( void ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

Remark See "Absolute address allocation specification (__directmap)".

(2) Using section names

Change the compiler output section name and specify a starting address.

Use the #pragma directive to specify the name of the section to be changed, a new name, and the starting address of the new section.

The following example changes the section name from @@CODEL to CC1, and specifies 2400H as the starting address.

```
#pragma section @@CODEL CC1 AT  2400H

void main ( void ) {
        /* Function definition */
}
```

Remarks 1. See "Changing compiler output section name (#pragma section ...)".

2.2.6 Defining 1-bit variables

The variable is made bit and boolean type, are handled as 1-bit data, and are allocated in the short direct addressing area.

bit and boolean type variables are handled in the same way as external variables with no initial values (irregularity).

The compiler generates the following bit manipulation instructions to this bit variables.

```
MOV1 , AND1 , OR1 , XOR1 , SET1 , CLR1 , NOT1 , BT , BF
```

The bit access to the short direct addressing area becomes possible in C source code.

```
#define ON      1
#define OFF     0
extern bit data1 ;
extern bit data2 ;

void main ( void ) {
        data1 = ON ;
        data2 = OFF ;
        while (data1 ) {
                data1 = data2 ;
                testb ( );
        }
        if ( data1 && data2 ) {
                chgb ( );
        }
}
```

Remark See "bit type variables (bit), boolean type variables (boolean/__boolean)".

2.2.7 Empty area of the structure is stuffed

Specify the -rc option to deselect alignment of structure members on 2-byte boundaries.

However, there is no support for deselecting alignment of non-structure variables.

2.3 Functions

This section explains functions.

2.3.1 Allocating to sections accessible with short instructions

Using callt function calls, obtain code that is more compact that the code for normal function calls.

A callt instruction stored the address of the called function in the area [80H - 0BFH] called a callt table. And possible to call the function by a short cord than the function is called directly.

```
__callt void func1 ( void );
__callt void func1 ( void ) {
            :
        /* Function definition */
            :
}
```

Remark See "callt functions (callt/__callt)".

2.3.2 Allocating in near areas

Using the small model, the compiler generates code with 16-bit address lengths.

When knowing in advance that code and data are into 64 KB, obtain more compact code by using the small model instead of the large model.

Specify the small model (-ms option) with the compiler option. Functions are allocated in near areas.

Or add the __near type qualifier to function declarations.

```
__near int func ( void );            /* Allocating in near area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__near int func ( void ) {           /* Allocating in near area */
        /* Function processing */
        return 0 ;
}

void main ( void ) {
        int     a;
        a = func ( ) ;
}
```

Remark Refer to "near/far area specification".

2.3.3 Allocating in far areas

If data are into 64 KB and code are into 1MB, use the medium model.

Specify the medium model (-mm option) with the compiler option. Functions are allocated in far areas.

If code and data are into 1 MB, use the large model.

Specify the large model (-ml option) with the compiler option. Data and functions are allocated in far areas.
Or add the __far type qualifier to function declarations.

```
__far  int func ( void );            /* Allocating in far area */
__near const int a = 0 x 12 ;        /* Allocating in near area */
__near int b = 0 x 12 ;              /* Allocating in near area */

__far int func ( void ) {            /* Allocating in far area */
        /* Function processing */
        return 0 ;
}

void main ( void ) {
        int     a;
        a = func ( ) ;
}
```

Remark Refer to "near/far area specification".

2.3.4 Allocating addresses directly

(1) Using section names

Change the compiler output section name and specify a starting address.

Use the #pragma directive to specify the name of the section to be changed, a new name, and the starting address of the new section.

```
#pragma section @@DATA  ??DATA  AT 0FDE00H
int             a1 ;                        // ??DATA
int             a2 ;                        // ??DATA
#pragma section @@DATS  ??DATS  AT 0FFE30H
sreg int        b1 ;                        // ??DATS
sreg int        b2 ;                        // ??DATS
```

Remark See "Changing compiler output section name (#pragma section ...)".

2.3.5 Inline expansion of function

#pragma inline instructs to generate inline expansion code for memory operation standard library memcpy and memse, instead of calling functions.

If to make the execution faster by expanding other functions inline, there are no instructions which can be inline expansive every function. If the function except memcpy and memset being inline-expansive, define a macro in function format,as shown below.

```
#define MEMCOPY ( a, b, c ) \
        { \
                struct st { unsigned char d[ ( c ) ]; }; \
                * ( ( struct st * ) ( a ) ) = * ( ( struct st * ) ( b ) ); \
        }
```

Remark See "Memory manipulation function (#pragma inline)".

2.3.6 Embedding assembly instructions

Embedding assembly instructions in the assembler source file output by the compiler.

(1) #asm - #endasm

#asm marks the start of an assembly source code block, and #endasm marks its end. Write assembly source code between the #asm and #endasm.

```
#asm
    :   /* Assembly source */
#endasm
```

[Output assemble file] by [Compile Options] tab of Property panel, set it as "Yes." (See the "CubeSuite 78K0R build" for a setting method.)

Remark See "ASM statements (#asm - #endasm/__asm)".

(2) __asm

Described by the next form in the C source.

```
__asm ( string literal );
```

Characters in the string literal are interpreted according to the ANSI conventions. Escape sequences, the line continues on the next line by '\' character, and concatenate strings can be described.
[Output assemble file] by [Compile Options] tab of Property panel, set it as "Yes." (See the "CubeSuite 78K0R build" for a setting method.)

Remark See "ASM statements (#asm - #endasm/__asm)".

2.4 Using Microcontroller Functions

This section explains using microcontroller functions.

2.4.1 Accessing special function registers (SFR) from C

(1) Setting each register of SFR

The SFR area are a area of group of special function registers, such as mode and control registers for the peripheral hardware of 78K microcontrollers (PM1, P1, TMC80, etc.).
To use the SFR area from C, place the #pragma sfr at the start of C source file. This declares the name of each SFR register. The sfr keyword can be either uppercase or lowercase.

```
#pragma sfr
```

The following error message appears if attempt to use the SFR area without declaring the register names.

```
E0711 Undeclared 'variable-name' ; function 'function-name'
```

The symbols made available by the #pragma sfr directive are the same as the abbreviations given in the list of special function registers.
The following items can be described before #pragma sfr:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

In the C source, simply use the sfr names supported by the target device. The sfr names do not need to be declared individually.
SFR names are external variables with no initial values (irregularity).
A compiler error occurs if assign invalid constant data to an SFR name.

Remark See "How to use the sfr area (sfr)".

(2) Specifying bits in SFR registers

As shown below, specify bits in SFR registers by using reserved names or by using the "register-name.bit-position".

Examples 1. Starting TM1

```
TCE1 = 1 ;
or
TMC1.0 = 1 ;
```

2. Stopping TM1

```
TCE1 = 0 ;
or
TMC1.0 = 0 ;
```

2.4.2 Interrupt functions in C

(1) Interrupt function

The following two directives are provided when the interrupt function is specified.

- #pragma interrupt
- #pragma vect

Either can be used. And the vector table is generated , which can check in the assembler source list output.

Place the #pragma directive at the start of the C source file.

The following items can be described before #pragma directives:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

Example Processing for input to INTP0 pin

```
#pragma interrupt INTP0 inter rb1
void inter ( void ) {
        /* Processing for input to INTP0 pin*/
}
```

Remark See " Interrupt functions (#pragma vect/#pragma interrupt)".

(2) RTOS interrupt handlers

RTOS interrupt handlers are described by use the #pragma rtos_interrupt, as shown below.

Place the #pragma directive at the start of the C source file.

The following items can be described before #pragma directives:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

```
#pragma rtos_interrupt INTP0 inthdr1

#include "kernel.h"
#include "kernel_id.h"

void inthdr1 ( void ) {
        /* Handle the interrupt */
        return ;
}
```

Remark Refer to " Interrupt handler for RTOS (#pragma rtos_interrupt ...)".

(3) Allocating stack area

When using the extended functions for interrupt functions, and do not specify stack switching, the compiler uses the default stack. It does not allocate any extra stack space that be required.

2.4.3 Using CPU control instructions in C

(1) halt instruction

The halt instruction is one of the standby functions of the microcontroller. To use it, use the #pragma HALT as shown below.

Place the #pragma directive at the start of the C source file.

The following items can be described before #pragma directives:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

In the form similar to the function call, describes it by the uppercase letter in C source as follows.

Example Using the halt instruction

```
#pragma HALT
    :

void func ( void ) {
    :
        HALT ( );
}
```

Remark See " CPU control instruction(#pragma HALT/STOP/BRK/NOP)".

(2) stop instruction

The stop instruction is one of the standby functions of the microcontroller. To use it, use the #pragma STOP as shown below.

Place the #pragma directive at the start of the C source file.

The following items can be described before #pragma directives:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

In the form similar to the function call, describes it by the uppercase letter in C source as follows.

Example Using the stop instruction

```
#pragma STOP
        :

void func ( void ) {
        :
        STOP ( );
}
```

Remark See " CPU control instruction(#pragma HALT/STOP/BRK/NOP)".

(3) brk instruction

To use software interrupt of a microcontroller, use the #pragma BRK as shown below.

Place the #pragma directive at the start of the C source file.

The following items can be described before #pragma directives:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

In the form similar to the function call, describes it by the uppercase letter in C source as follows.

Example Using the brk instruction

```
#pragma BRK
        :

void func ( void ) {
        :
        BRK ( );
}
```

Remark See " CPU control instruction(#pragma HALT/STOP/BRK/NOP)".

(4) nop instruction

The nop instruction advances the clock without operating a microcontroller. To use it, use the #pragma NOP as shown below.

Place the #pragma directive at the start of the C source file.

The following items can be described before #pragma directives:

- Comments
- Preprocessor directive which does neither define nor refer to a variable or a function

In the form similar to the function call, describes it by the uppercase letter in C source as follows.

Example Using the nop instruction

```
#pragma NOP
        :

void func ( void ) {
        :
        NOP ( );
}
```

Remark See " CPU control instruction(#pragma HALT/STOP/BRK/NOP)".

2.5 Startup Routine

This section explains startup routin.

2.5.1 Deleting unused functions and areas from startup routine

(1) Deleting the exit function

Delete the exit function by setting the EQU symbol EXITSW in the startup routine to 0.

(2) Deleting unused areas

An unused area about the area such as _ @FNCTBL that a standard library uses can be deleted by confirming the library used, and changing the value of the EQU symbol such as EXITSW in startup routine cstart.asm. The following table lists the controlling EQU symbols and the affected library function names and symbol names.

EQU Symbol	Library Function Name	Symbol Name
BRKSW	brk sbrk malloc calloc realloc free	_errno _@MEMTOP _@MEMBTM _@BRKADR
EXITSW	exit	_@FNCTBL _@FNCENT
RANDSW	rand srand	_@SEED
DIVSW	div	_@DIVR
LDIVSW	ldiv	_@LDIVR
STRTOKSW	strtok	_@TOKPTR
FLOATSW	vatof strtod Math functions Floating point runtime library	_errno

Remark See "7.4 Startup Routines".

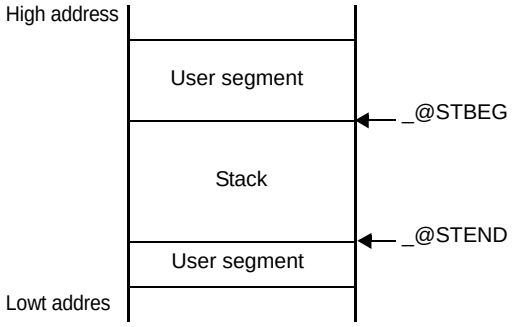
2.5.2 Allocating stack area

(1) Stack setting

If specify the stack resolution symbol option -s when linking, the symbol _@STEND is generated to mark the lowest address in the stack, and the symbol _@STBEG is generated to mark the highest address + 1.

```
-sSTACK    <-- Stack area defined by directive
```

Figure 2-1. Stack Setting



In this case, set the stack pointer as follows.

```
MOVW    SP, #LOWW       _@STBEG
```

(2) Checking stack area

To check the stack area, specify the linker -kp option to output the public symbol list in the link list file.
The stack area is between the _@STEND symbol and the _@STBEG symbol.

Example Public symbol list

```
*** Public symbol list ***

MODULE  ATTR    VALUE   NAME

        NUM     0FFE20H  _@STBEG
        NUM     0FFB7EH  _@STEND
```

2.5.3 Initializing RAM

In the default startup routine, initial values are copied to the following areas.

- @@INIT segment
- @@INITL segment
- @@INIS segment

The following areas are zero cleared.

- saddr area (0FFE20H to 0FFEDFH)
- @@DATA segment
- @@DATAL segment
- @@DATS segment

If to initialize areas other than the above, add the appropriate initialization processing code to the startup routine.

Remark See "7.4 Startup Routines".

2.6 Link Directives

This section explains link directives.

2.6.1 Partitioning default areas

Link directives allow to specify names for memory areas that define. However, care is required regarding the location of the special function register (SFR) area.

For example, if define two areas in RAM and specify 1) the name "RAM", which is defined by default, and 2) the user-defined name "STACK", then should make sure that the SFR area is contained within the area named RAM.

Example Link directives

```
MEMORY  STACK : ( 0FEF00H , 00100H )
MEMORY  RAM :   ( 0FF000H , 01000H )
```

Remark See "5.1.1 Link directives".

2.6.2 Specifying section allocation

(1) Specifying areas

When specifying the allocation of a section, can specify a memory area.

Use the MERGE quasi directive to allocate the target section in a memory area.

Example Allocate input segment SEG1 to memory area MEM1.

```
MEMORY  ROM : ( 0000H , 1000H )
MEMORY  MEM1 : ( 1000H , 2000H )
MERGE   SEG1 : = MEM1
```

Remark See "5.1.1 Link directives".

(2) Specifying addresses

When specifying the allocation of a section, can specify addresses.

Use the MERGE quasi directive to specify the allocation address of the target section.

Example Allocate input segment SEG1 to address 500H.

```
MEMORY  ROM : ( 0000H , 10000H )
MERGE   SEG1 : AT ( 500H )
```

Remark See "5.1.1 Link directives".

2.7 Reducing Code Size

This section explains techniques for reducing the code size.

2.7.1 Using extended functions to generate efficient object code

When 78K0R application product is developed, 78K0R C compiler generates efficient object code by using the saddr and callt areas in the device.

- Using external variables
  - if (saddr area available)
    - use sreg/__sreg variables/ or compiler's -rd option
- Using 1 bit data
  - if (saddr area available)
    - use bit/boolean/__boolean type variables
- Function definitions
  - if (frequently called function)
    - if (callt area available)
      - define as __callt/callt function (for smaller code size)
  - if (use automatic variables && saddr area available)

(1) Using external variables

If available in the saddr area when defineing external variables, define external variables as sreg/__sreg variables.

sreg/__sreg variables are shorter instruction code than the instructions to memory. Object code will be smaller and execution speed will be faster. (Instead of the sreg variables, can use the compiler -rd option.)

sreg/__sreg variable define :	`extern sreg int` *variable-name* ;
	`extern __sreg int` *variable-name* ;

Remark See "How to use the saddr area (sreg/__sreg)".

(2) Using 1 bit data

When using only 1 bit of data, define a bit type (or boolean/__boolean type) variable. The compiler generates bit operation instructions to manipulate bit/boolean/__boolean type variables. Like sreg variables, they are stored in the saddr area for smaller code and faster execution speed.

bit/boolean type variable define :	`bit`	*variable-name* ;
	`boolean`	*variable-name* ;
	`__boolean`	*variable-name* ;

Remark See "bit type variables (bit), boolean type variables (boolean/__boolean)".

(3) Function definitions

Frequently called functions can be registered in the callt table when callt area can be used.
callt functions are called by using the callt areas of the device, so they can be called by code that is shorter than normal function calls.

```
callt function define : callt   int     tsub ( ) {
                              :
                        }
```

Remark See "callt functions (callt/__callt)".

Without changing the C source code by the thing compiled using optimization option in addition to use in saddr area, it's possible to generate a high-quality object.

(4) Using extended functions

- Function definitions
  - if (use automatic variables && saddr area available)
    - Register definitions
  - if (use internal static variable) && (saddr area available)
    - __sreg definitions

(a) Functions that use automatic variables

When the function for which an automatic variable is used can use saddr area, define register. A register definitions allocates a defined object to a register.
Programs that use registers are shorter object and faster execution than programs that use memory.

Remark About defining register variables (register int i ; ...), see "Register variables (register)".

(b) Functions that use internal static variables

When the function for which an internal static variables is used can use saddr area, define __sreg or specify the -rs option. Like sreg variables, they are possible to shorter object and faster execution.

Remark See "How to use the saddr area (sreg/__sreg)".

(5) Other functions

Other extended functions allow to generate faster execution or more compact code.

(a) Use SFR names (or SFR bit names)

#pragma sfr

Remark See "How to use the sfr area (sfr)".

(b) __sreg definitions for bit fields of 1-bit members (members can also use unsigned char type)

```
__sreg  struct  bf {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
} bf_1 ;
```

Remark See "How to use the saddr area (sreg/__sreg)".

(c) Use register bank switching for interrupt routines

#pragma interrupt INTP0 inter RB1

Remark See "Interrupt functions (#pragma vect/#pragma interrupt)".

(d) Use of multiplication, division embedded function

#pragma mul
#pragma div

Remark See "Multiplication function (#pragma mul)", Division function (#pragma div)".

(e) Described by assembly language to be faster modules.

2.7.2 Calculating complex expressions

The following example shows the most reasonable way to calculate an expression whose result will always fit into byte type, even when intermediate results require double word type.

Example Find the rounded percentage C of B in A.

```
C = ( A x 100 + B ÷ 2 ) ÷ B
```

In a function like the following, the variable for the result C must be defined as a long int, requiring 4 bytes of area when a single byte would have been enough.

```
void _x ( ) {
        c = ( ( unsigned long int ) a * ( unsigned long int ) 100 + ( unsigned long int
) b / ( unsigned long int ) 2 ) / ( unsigned long int ) b ;
}
```

This can be written as follows, if using double word type for intermediate results only.

```
#pragma mul
#pragma div
unsigned int    a, b ;
unsigned char   c ;

void _x ( ) {
       c = ( unsigned char ) divux ( ( unsigned long ) ( b / 2 ) + muluw ( a, 100 ) , b
);
}
```

2.8 Compiler and Assembler Mutual References

This section explains compiler and assembler mutual references.

2.8.1 Mutually referencing variables

(1) Reference a variable defined in C language

To reference a extern variable defined in a C program from an assembly language routine, define extern. Prefix the name of the variable with an underscore (_) in the assembly language module.

Example C source

```
extern  void subf ( void );
char    c = 0 ;
int     i = 0 ;

void main ( void ) {
        subf ( );
}
```

Example Assembly source

```
$PROCESSOR ( F1166A0 )
        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i
@@CODE CSEG
_subf :
        MOV     !_c, #04H
        MOVW    AX, #07H
        MOVW    !_i, AX
        RET
        END
```

Remark See "9.5 Referencing Variables Defined in C Language".

(2) Reference a variable defined in assembly language

To reference a extern variable defined in an assembly language program from a C routine, define extern. Prefix the name of the variable with an underscore (_) in the assembly language routine.

Example C source

```
extern char     c ;
extern int      i ;

void subf ( void ) {
        c = ' A ' ;
        i = 4 ;
}
```

Example Assembly source

```
NAME ASMSUB
                PUBLIC  _i
                PUBLIC  _c
ABC DSEG        BASEP
_i : DW         0
_c : DB         0
                END
```

Remark See "9.6 Referencing Variables Definted in Assembly Language from C Language".

2.8.2 Mutually referencing functions

(1) Reference a function defined in C language

The following procedure is used to call functions written in C from assembly language routines.

(a) Save the C work registers (AX, BC, DE)

(b) Push the arguments on the stack

(c) Call the C function

(d) Adjust the stack pointer (SP) by the byte length of the arguments

(e) Reference the return value of the C function (BC, or DE, BC)

Example Assembly language

```
$PROCESSOR ( F1166A0 )
        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2
@@CODE CSEG
_FUNC2 :
        movw    ax, #20H        ; set 2nd argument ( j )
        push    ax              ;
        movw    ax, #21H        ; set 1st argument ( i )
        call    !_CSUB          ; call "CSUB ( i, j )"
        pop     ax              ;
        ret
END
```

See "9.4 Calling C Language Routines from Assembly Language".

(2) Reference a function defined in assembly language

Functions defined in assembly language to be called from C functions perform the following processing.

(a) Save the base pointer and saddr area for register variables

(b) Copy the stack pointer (SP) to the base pointer (HL)

(c) Perform the processing of the function FUNC

(d) Set the return value

(e) Restore the saved registers

(f) Return to the function main

Example Assembly language

```
$PROCESSOR ( F1166A0 )
        PUBLIC _FUNC
        PUBLIC _DT1
        PUBLIC _DT2
@@DATA DSEG BASEP
_DT1 : DS ( 2 )
_DT2 : DS ( 4 )
@@CODE CSEG
_FUNC :
        PUSH    HL              ; save base pointer
        PUSH    AX
        MOVW    HL, SP          ; copy stack pointer
        MOVW    AX, [HL]        ; arg1
        MOVW    !_DT1, AX       ; move 1st argument ( i )
        MOVW    AX, [HL + 10]   ; arg2
        MOVW    !_DT2 + 2, AX
        MOVW    AX, [HL + 8]    ; arg2
        MOVW    !_DT2, AX       ; move 2nd argument ( l )
        MOVW    BC, #0AH        ; set return value
        POP     AX
        POP     HL              ; restore base pointer
        RET                     ;
        END
```

See "9.3 Calling Assembly Language Routines from C Language".

CHAPTER 3 COMPILER LANGUAGE SPECIFICATIONS

This chapter explains the language specifications supported by 78K0R C compiler.

3.1 Basic Language Specifications

The C compiler supports the language specifications stipulated by the ANSI standards. These specifications include items that are stipulated as processing definitions. This chapter explains the language specifications of the items dependent on the processing system of the micro processors for 78K0R microcontrollers.

The differences between when options strictly conforming to the ANSI standards are used and when those options are not used are also explained.

See "3.3 Extended Language Specifications" for extended language specifications explicitly added by 78K0R C compiler.

3.1.1 Processing system dependent items

This section explains items dependent on processing system in the ANSI standards.

(1) Data types and sizes

The byte order in multibyte data types is "from least significant to most significant byte" Signed integers are expressed by 2's complements. The sign is added to the most significant bit (0 for positive or 0, and 1 for negative).

- The number of bits of 1 byte is 8.
- The number of bytes, byte order, and encoding in an object files are stipulated below.

Table 3-1. Data types and Sizes

Data Types	Sizes
char	1 byte
int, short	2 bytes
long, float, double	4 bytes
pointer	near : 2 bytes far : 4 bytes

(2) Translation stages

The ANSI standards specify eight translation stages (known as "phases") of priorities among syntax rules for translation. The arrangement of "non-empty white space characters excluding line feed characters" which is defined as processing system dependent in phase 3 "Decomposition of source file into preprocessing tokens and white space characters" is maintained as it is without being replaced by single white space character. However, tabs are replaced by the space character specified with the -lt option.

(3) Diagnostic messages

When syntax rule violation or restriction violation occurs on a translation unit, the compiler outputs as error message containing source file name and (when it can be determined) the number of line containing the error. These error messages are classified into three types: "alarm", "fatal error", and "other error" messages.

(4) Free standing environment

(a) The name and type of a function that is called on starting program processing are not stipulted in a free-standing environment[Note]. Therefore, it is dependent on the user-own coding and target system.

Note Environment in which a C Language source program is executed without using the functions of the operating system.
In the ANSI Standard two environments are stipulated for execution environment: a free-standing environment and a host environment. The 78K0R C compiler does not supply a host environment at present.

(b) The effect of terminating a program in a free-standing environment is not stipulated. Therefore, it is dependent on the user-own coding and target system.

(5) Program execution

The configuration of the interactive unit is not stipulated.
Therefore, it is dependent on the user-own coding and target system.

(6) Character set

The values of elements of the execution environment character set are ASCII codes.

(7) Multi-byte characters

Multi-byte characters are not supported by character constants and character strings.
However, Japanese description in comments is supported.

(8) Significance of character display

The values of expanded notation are stipulated as follows.

Table 3-2. Expanded Notation and Meaning

Expanded Notation	Value (ASCII)	Meaning
\a	07	Alert (Warning tone)
\b	08	Backspace
\f	0C	Form feed (New Page)
\n	0A	New line (Line feed)
\r	0D	Carriage return (Restore)
\t	09	Horizontal tab
\v	0B	Vertical tab

(9) Translation Limit

The limit values of translation are explained below.

The values marked with * are guaranteed values. These values may be exceeded in some cases, but the operation is not guaranteed.

Table 3-3. Translation Limit Values

Contents	Limit Values
Number of nesting levels of compound statements, repetitive control structures, and selective control structures (However, dependent on the number of "case" labels)	45
Number of nesting levels of condition embedding	255
Number of pointers, arrays, and function declarators (in any combination) qualifying one arithmetic type, structure type, union type, or incomplete type in one declaration	12
Number of nesting levels enclosed by parentheses in a complete declarator	591*
Number of nesting levels of an expression enclosed by parentheses in a complete expression	32
Valid number of first characters in a macro name	256
Valid number of first characters of an external identifier	249
Valid number of first characters in an internal identifier	249
Number of identifiers having an external identifier in one translation unit	1024*
Number of identifiers having the valid block range declared in one basic block	255
Number of macro identifiers simultaneously defined in one translation unit	32767
Number of dummy arguments in one function definition and number of actual arguments in one function call	39*
Number of dummy arguments in one macro definition	31
Number of actual arguments in one macro call	31
Number of characters in one logical source line	2048*
One character string constant after concatenation, or number of characters in a wide character string constant	509*
Object size of 1-file (Data is indicated)	65535
Number of nesting levels for include (#include) files	50
Number of "case" labels for one "switch" statement (including those nested, if any)	257
Number of source lines per compilation unit	65535*
Number of nested function calls	40*
Total size of code, data, and stack segments in a single object module	by memory model[Note]
Number of members of a single structure or single union	256
Number of enumerate constants in a single enumerate type	255
Number of nesting levels of a structure or union definition in the arrangement of a single structure declaration	15
Nesting of initializer elements	15
Number of function definitions in a single source file	4095
Macro nesting	200
Number of include file paths	64

Note The following table lists the maxium values for each memory model when extended functions are not used.

Memory Model	Maximum Values
Small model	Code 64KB, Data 64KB, Total 128KB
Medium model	Code 1MB, Data 64KB, Total 1MB
Large model	Code 1MB, Data 1MB, Total 1MB

(10) Quantitative limit

(a) The limit values of the general integer types (limits.h file)

The limits.h file specifies the limit values of the values that can be expressed as general integer types (char type, signed/unsigned integer type, and enumerate type).

Because multibyte characters are not supported, MB_LEN_MAX does not have a corresponding limit. Consequently, it is only defined with MB_LEN_MAX as 1.

If a -qu option is specified, CHAR_MIN is 0, and CHAR_MAX takes the same value as UCHAR_MAX. The limit values defined by the limits.h file are as follows.

Table 3-4. Limit Values of General Integer Type (limits.h File)

Name	Value	Meaning
CHAR_BIT	+8	The number of bits (= 1 byte) of the minimum object not in bit field
SCHAR_MIN	-128	Minimum value of signed char
SCHAR_MAX	+127	Maximum value of signed char
UCHAR_MAX	+255	Maximum value of unsigned char
CHAR_MIN	-128	Minimum value of char
CHAR_MAX	+127	Maximum value of char
SHRT_MIN	-32768	Minimum value of short int
SHRT_MAX	+32767	Maximum value of short int
USHRT_MAX	+65535	Maximum value of unsigned short int
INT_MIN	-32768	Minimum value of int
INT_MAX	+32767	Maximum value of int
UINT_MAX	+65535	Maximum value of unsigned int
LONG_MIN	-2147483648	Minimum value of long int
LONG_MAX	+2147483647	Maximum value of long int
ULONG_MAX	+4294967295	Maximum value of unsigned long int

(b) The limit values of the floating-point type (float.h file)

The limit values related to characteristics of the floating-point type are defined in float.h file. The limit values defined by the float.h file are as follows.

Table 3-5. Definition of Limit Values of Floating-point type (float.h File)

Name	Value	Meaning
FLT_ROUNDS	+1	Rounding mode for floating-point addition. 1 for the 78K0R microcontrollers (rounding in the nearest direction).
FLT_RADIX	+2	Radix of exponent (b)
FLT_MANT_DIG	+24	Number of numerals (p) with FLT_RADIX of floating- point mantissa as base
DBL_MANT_DIG		
LDBL_MANT_DIG		
FLT_DIG	+6	Number of digits of a decimal number[Note 1] (q) that can round a decimal number of q digits to a floating-point number of p digits of the radix b and then restore the decimal number of q
DBL_DIG		
LDBL_DIG		
FLT_MIN_EXP	-125	Minimum negative integer (e_{min}) that is a normalized floating-point number when FLT_RADIX is raised to the power of the value of FLT_RADIX minus 1.
DBL_MIN_EXP		
LDBL_MIN_EXP		
FLT_MIN_10_EXP	-37	Minimum negative integer$\log_{10} b^{e_{min-1}}$ that falls in the range of a normalized floating-point number when 10 is raised to the power of its value.
DBL_MIN_10_EXP		
LDBL_MIN_10_EXP		
FLT_MAX_EXP	+128	Maximum integer (e_{max}) that is a finite floating-point number that can be expressed when FLT_RADIX is raised to the power of its value minus 1.
DBL_MAX_EXP		
LDBL_MAX_EXP		
FLT_MAX_10_EXP	+38	Maximum value of finite floating-point numbers that can be expressed $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX_10_EXP		
LDBL_MAX_10_EXP		
FLT_MAX	3.40282347E + 38F	Maximum value of finite floating-point numbers that can be expressed $(1 - b^{-p}) * b^{e_{max}}$
DBL_MAX		
LDBL_MAX		
FLT_EPSILON	1.19209290E - 07F	Difference[Note 2] between 1.0 that can be expressed by specified floating-point number type and the lowest value which is greater than 1. b^{1-p}
DBL_EPSILON		
LDBL_EPSILON		
FLT_MIN	1.17549435E - 38F	Minimum value of normalized positive floating-point number $b^{e_{min-1}}$
DBL_MIN		
LDBL_MIN		

Notes 1. DBL_DIG and LDBL_DIG are 10 or more in the ANSI standards but are 6 in the 78K0R microcontrollers because both the double and long double types are 32 bits.

2. DBL_EPSILON and LDBL_EPSILON are 1E-9 or less in the ANSI standards, but 1.19209290E-07F in the 78K0R microcontrollers.

(11) Identifier

The initial 249 characters of identifiers are recognized.
Uppercase and lowercase characters are distinguished.

(12) char type

A char type with no type specifier (signed, unsigned) specified is treated as a signed integer as the default assumption.
However, a simple char type can be treated as an unsigned integer by specifying the - qi option of the C compiler.
The types of those that are not included in the character set of the source program required by the ANSI standards (escape sequence) is converted for storage, in the same manner as when types other than char type are substituted for a char type.

```
char    c = ’ \777’ ;     /* Value of c is -1 */
```

(13) Floating-point constants

The floating-point constants conform to IEEE754[Note].

Note IEEE**:**Institute of Electrical and Electronics Engineers
Moreover,IEEE754 is a standard to unify specifications such as the data format and numeric range in systems that handle floating-point operations.

(14) Character constants

(a) Both the character set of the source program and the character set in the execution environment are basically ASCII codes, and correspond to members having the same value.

(b) The last character of the value of an integer character constant including two or more characters is valid.

(c) A character that cannot be expressed by the basic execution environment character set or escape sequence is expressed as follows.

<1> An octal or hexadecimal escape sequence takes the value indicated by the octal or hexadecimal notation

\077	63

<2> The simple escape sequence is expressed as follows.

\'	'
\"	"
\?	?
\\	\

<3> Values of \a, \b, \f, \n, \r, \t, \v are same as the values explained in "(8) Significance of character display".

(d) Character constants of multi byte characters are not supported.

(15) Header file name

The method to reflect the string in the two formats (< > and " ") of a header file name on the header file or an external source file name is stipulated in "(32) Loading header file".

(16) Comment

A comment can be described in Japanese. The default character code set for Japanese is Shift JIS.
The character code set of the input source file can be specified by the compiler's -z option, or by an environmental variable. An option specification takes priority over an environment variable specification.
However, character codes are not guaranteed when "none" is specified.

(a) Option specification

```
-ze | -zn | -zs
```

(b) Environment variable

```
LANG78K [ euc | none | sjis ]
```

To set environment variables, use the standard procedure for environment.

(17) Signed constants and unsigned constants

If the value of a general integer type is converted into a signed integer of a smaller size, the higher bits are truncated and a bit string image is copied.
If an unsigned integer is converted into the corresponding signed integer, the internal representation is not changed.

(18) Floating-points and general integers

If the value of a general integer type is converted into the value of a floating-point type, and if the value to be converted is within a range that can be expressed but not accurately , the result is rounded to the closest expressible value.
When the result is just a middle value, it can be rounded to the even number (with the least significant bit of the mantissa being 0).

(19) double type and float type

In the 78K0R C compiler, a double type is expressed as a floating-point number in the same manner as a float type, and is treated as 32-bit (single-precision) data

(20) Signed type in operator in bit units

The characteristics of the shift operator conform to the stipulation in"(26) Shift operator in bit units" .
The other operators in bit units for signed type are calculated as unsigned values (as in the bit imag.

(21) Members of structures and unions

If the value of a member of a union is stored in a different member, it is stored according to an alignment condition. Therefore, the members of that union are accessed according to the alignment condition (see "(b) Structure type" and "(c) Union type").
In the case of a union that includes a structure sharing the arrangement of the common first members as a member, the internal representation is the same, and the result is the same even if the first member common to any structure is referred.

(22) sizeof operator

The value resulting from the "sizeof" operator conforms to the stipulation related to the bytes in an object in"(1) Data types and sizes".

For the number of bytes in a structure and union, it is byte including padding area.

(23) Cast operator

When a pointer is converted into a general integer type, the required size of the variable is the same as the following table lists. The bit string is saved as is as the conversion result.

Any integer can be converted by a pointer. However, the result of converting an integer smaller than an int type is expanded according to the type.

near	2 bytes
far	4 bytes

When a near pointer or int is cast to a far pointer, and when a near pointer is cast to a long, the operation behaves as follows.

- For variable pointers, 0xf is added at the most significant position. (0 is an exception. The pointer is zero-extended.)
- Function pointers are zero-extended.

(24) Division/remainder operator

The result of the division operator ("/") when the operands are negative and do not divide perfectly with integer division, is as follows: If either the divisor or the dividend is negative, the result is the smallest integer greater than the algebraic quotient.

If both the divisor and the dividend are negative, the result is the largest integer less than the algebraic quotient.

If the operand is negative, the result of the "%" operator takes the sign of the first operand in the expression.

(25) Addition and subtraction operators

If two pointers indicating the elements of the same array are subtracted, the type of the result is int type, and the size is 2 bytes.

(26) Shift operator in bit units

If E1 of "E1 >> E2" is of signed type and takes a negative value, an arithmetic shift is executed.

(27) Storage area class specifier

The storage area class specifier "register" is declared to increase the access speed as much as possible, but this is not always effective.

(28) Structure and union specifier

(a) int type bit field sign Simple int type bit fields without a signed or unsigned specifier are treated as unsigned.

(b) To retain a bit field, a storage area unit to which any address with sufficient size can be assigned can be allocated. If there is insufficient area, however, the bit field that does not match is packed into to the next unit according to the alignment condition of the type of the field.

(c) The allocation sequence of the bit field in unit is from lower to higher.
However, the -rb option can be specified for the allocation sequence is from higher to lower.

(d) Each member of the non-bit field of one structure or union is aligned at a boundary as follows

cchar and unsigned char types, and arrays of char and unsigned char types	Byte boundary
Other (including pointers)	2-byte boundary

(29) Enumerate type specifier

The type of an enumeration is the first type from among the following which is capable of expressing all of the enumeration constants.

- signed char
- unsigned char
- signed int

(30) Type qualifier

The configuration of access to data having a type qualified to be "volatile" is dependent upon the address (I/O port, etc.) to which the data is mapped.

(31) Condition embedding

(a) The value for the constant specified for condition embedding and the value of the character constant appearing in the other expressions are equal.

(b) The character constant of a single character must not have a negative value.

(32) Loading header file

(a) A preprocessing directive in the form of "#include <character string>"

Unless "filename" begins with the character '\' [Note] the #include <filename> preprocessor directive instructs the preprocessor to search for the file specified between the angle brackets (<..>) in the following locations: 1) the folder specified by the -i option, 2) the folder specified by the INC78K0R environment variable, and 3) the ..\inc78k0r folder relative to the bin folder where cc78k0r.exe resides.

If a header file uniformly identified is searched with a character string specified between delimiters "<" and ">", the whole contents of the header file are replaced.

Note Both "\" and "/" are regarded as the delimiters of a folder.

Example

```
#include        <header.h>
```

The search order is as follows.

- The folder specified by the -i option
- The folder specified by the INC78K0R environment variable
- The standard folder

(b) A preprocessing directive in the form of "#include "character string""

Unless "character string" begins with the character '\'[Note], the #include "character string" preprocessor directive instructs the preprocessor to search for the file specified between the quotation marks ("..") in the following locations: 1) the folder that contains the source file, 2) the folder specified by the -i option, 3) the folder specified by the INC78K0R environment variable, and 4) the ..\inc78k0r folder relative to the bin folder where cc78k0r.exe resides.

If the file specified between the quotation mark delimiters is found, the #include directive line is replaced with the entire contents of the file.

Note Both "\" and "/" are regarded as the delimiters of a folder.

Example

```
#include        "header.h"
```

The search order is as follows.

- The folder that contains the source file
- The folder specified by the -i option
- The folder specified by the INC78K0R environment variable
- The standard folder

(c) The format of "#include preprocessing character phrase string"

The format of "#include preprocessing character phrase string" is treated as the preprocessing character phrase of single header file only if the preprocessing character phrase string is a macro that is replaced to the form of <character string> or "character string".

(d) A preprocessing directive in the form of "#include <character string>"

Between a string delimited (finally) and a header file name, the length of the alphabetic characters in the strings is identified,

And the file name length valid in the compiler operating environment is valid.

The folder that searches a file conforms to the above stipulation.

(33) #pragma directive

#pragma directives are one of the preprocessing directive types defined by the ANSI standard. The string that follows #pragma the compiler to translate in an implementation-defined manner.

When a #pragma directive is not recognized by the compiler, it is ignored and translation continues. If the directive adds a keyword, then an error occurs if the C source contains that keyword. To avoid the error, delete the keyword from the source or exclude it with #ifdef.

(34) Predefined macro names

All the following macro names are supported.

Macros not ending with "_ _ " are supplied for the sake of former C language specifications (K&R specifications). To perform processing strictly conforming to the ANSI standards, use macros with "_ _ " before and after.

Table 3-6. List of Supported Macros

Macro Name	Definition
__LINE__	Line number of source line at that point (decimal).
__FILE__	Name of assumed source file (character string constant).
__DATE__	Date of translating source file (character string constant in the form of "Mmm dd yyyy"). Here, the name of the month is the same as that created by the asctime function stipulated by ANSI standards (3 alphabetic characters with only the first character is capital letter) (The first character of dd is blank if its value is less than 10).
__TIME__	Translation time of source file (character string constant having format "hh:mm:ss" similar to the time created by the asctime function).
__STDC__	Decimal constant 1, indicating conformance to the ANSI standard.**Note**
__K0R__	Decimal constant 1
__K0R_SMALL__	Decimal constant 1 (When small model is specified.)
__K0R_MEDIUM__	Decimal constant 1 (When large model is specified.)
__K0R_LARGE__	Decimal constant 1 (When the -qu option was specified.)
__CHAR_UNSIGNED__	Decimal constant 1 (When the -qu option was specified.)
CPUmacro	Decimal constant 1 of a macro indicating the target CPU. A character string indicated by "product type specification" in the device file with "_ _ " prefixed and suffixed is defined.

Note Defined when the -za option is specified

(35) Definition of special data type

NULL, size_t, and ptrdiff_t defined by stddef.h file are as follows.

Table 3-7. Definition of NULL, size_t, ptrdiff_t (stddef.h File)

NULL/size_t/ptrdiff_t	Definition
NULL	((void *) 0)
size_t	unsigned int
ptrdiff_t	int

3.2 Environment During Compilation

This section explains how the C compiler handles data, registers, device specifications and the environment during execution.

3.2.1 Internal representation and value area of data

This section explains the internal representation and value area of each type for the data handled by the 78K0R C compiler.

(1) Basic types

The basic types, also called arithmetic types, consist of the integer types and the floating point types.
The integer types can be classified into the char type, signed integer types, unsigned integer types, and enumeration type.

(a) Integer types

Integer types can be divided into 4 categories, as follows. Integer types are expressed as binary 0s and 1s.

- char type
- signed integer types
- unsigned integer types
- enumeration types

<1> char type

The char type is large enough to store any member of the execution character set.
If a member of the basic execution character set is stored in a char object, its value is guaranteed to be nonnegative.
Objects other than characters are treated as signed integers.
If an overflow occurs when a value is stored, the overflow part is ignored.

<2> Signed integer types

There are four signed integer types, as follows.

- signed char
- short int
- int
- long int

An object defined as signed char type occupies the same amount of area as a "plain" char.
A "plain" int has the natural size suggested by the CPU architecture of the execution environment.
For each of the signed integer types, there is a corresponding unsigned integer type that uses the same amount of area.
The positive number of a signed integer type is a subset of the the unsigned integer type.

<3> Unsigned integer types

Unsigned integer types are designated by the keyword "unsigned".
A computation involving unsigned integer types can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modul the number that is one greater than the largest value that can be represented by the resulting type.

<4> Enumeration types

An enumeration comprises a set of named integer constant values.
Each distinct enumeration constitutes a different enumerated type.
Each enumeration constitutes a enumerated type.

(b) Floating point types

There are three real floating types, as follows.

- float
- double
- long double

Like the float type, the double and long double types of 78K0R C compiler are supported as floating point representations of the single-precision normalized numbers defined in ANSI/IEEE 754-1985. This means that the float, double, and long double types have the same value range.

Table 3-8. Value Ranges by Type

Type	Value Range
(signed) char	-128 to +127
unsigned char	0 to 255
(signed) short int	-32768 to +32767
unsigned short int	0 to 65535
(signed) int	-32768 to +32767
unsigned int	0 to 65535
(signed) long int	-2147483648 to +2147483647
unsigned long int	0 to 4294967295
float	1.17549435E - 38F to 3.40282347E + 38F
double	1.17549435E - 38F to 3.40282347E + 38F
long double	1.17549435E - 38F to 3.40282347E + 38F

Remarks 1. The "signed" type specifier may be omitted. However, when it is omitted for the char type, a compiling condition (option) determines whether the type is the signed char or unsigned char type.

2. The short int and int types have the same value range, but they are treated as different types.

3. The unsigned short int and unsigned int types have the same value range, but they are treated as different types.

4. The float, double, and long double types have the same value range, but they are treated as different types.

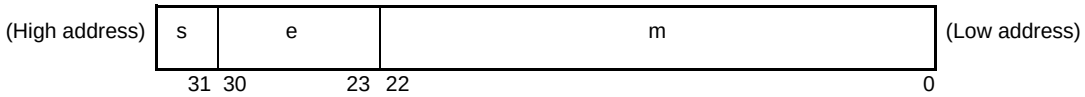
5. The ranges of the float, double, and long double types are ranges of absolute values.

The following show the specifications of floating point numbers (float type).

<1> Format

The floating point number format is shown below.

Figure 3-1. Floating Point Number Format



Numerical values in this format are as follows.

```
        (Value of sign)           (Value of exponent)
( -1 )              * (Value of mantissa) * 2
```

s	Sign (1 bit) 0 for a positive number and 1 for a negative number.
e	Exponent (8 bits) A base-2 exponent is expressed as a 1-byte integer (expressed by 2's complement in the case of a negative), after the further addition of a bias of 7FH. These relationships are shown in the table below. Exponent (Hexadecimal) / Value of Exponent FE / 127 : / : 81 / 2 80 / 1 7F / 0 7E / -1 : / : 01 / -126
m	Mantissa (23 bits) The mantissa is expressed as an absolute value, with bit positions 22 to 0 equivalent to the 1st to 23rd places of a binary number. Except for when the value of the floating point is 0, the value of the exponent is always adjusted so that the mantissa is within the range of 1 to 2 (normalization). The result is that the position of 1 (i.e. the value of 1) is always 1, and is thus represented by omission in this format.

<2> Expression of zero

When exponent = 0 and mantissa = 0, $\pm$ 0 is expressed as follows.

```
        (Value of sign)
( -1 )              * 0
```

<3> Expression of infinity

When exponent = FFH and mantissa = 0, $\pm \infty$ is expressed as follows.

```
        (Value of sign)
( -1 )              * ∞
```

<4> Denormalized values

When exponent = 0 and mantissa $\neq$ 0, the denormalized value is expressed as follows.

```
        (Value of sign)                -126
( -1 )              * (Value of mantissa) * 2
```

Remark The mantissa value here is a number less than 1, so bit positions 22 to 0 of the mantissa express the 1st to 23rd decimal places.

<5> Expression of NaN (Not-a-number)

When exponent = FFH and mantissa ≠ 0, NaN is expressed, regardless of the sign.

<6> Rounding of computation results

Numerical values are rounded down to the nearest even number. If the computation result cannot be expressed in the above floating point format, round to the nearest expressible number.
If there are 2 values that can express the differential of the prerounded value, round to an even number (a number whose least significant binary bit is 0).

<7> Exceptions

There are 5 types of exceptions, as shown in the table below.

Table 3-9. Numerical Exception

Exception	Return Value
Underflow	Denormalized number
Inexact	± 0
Overflow	± ∞
Division by zero	± ∞
Invalid operation	NaN

When an exception occurs, calling the matherr function causes a warning to appear.

(2) Character types

There are 3 char data types.

- char
- signed char
- unsigned char

(3) Incomplete types

There are 4 incomplete data types.

- Arrays with indefinite object size
- Structures
- Unions
- void type

(4) Derived types

There are 5 derived data types.

- Array type
- Structure type
- Union type
- Function type
- Pointer type

(a) Array type

An array type describes a contiguously allocated set of objects with a particular member object type, called the element type.
All member objects have the area of the same size. Array types and individual elements can be specified. It is not possible to create an incomplete array type.

(b) Structure type

A structure type describes a sequentially allocated set of member objects, each of which has an optionally specified name and possibly a distinct type.

Remark Array and structure types are collectively called aggregate types. The member objects in aggregate types are allocated sequentially.

(c) Union type

A union type describes an overlapping set of member objects.
Each member of a union has an optionally specified name and possibly a distinct type. Union members can be specified individually.

(d) Function type

A function type describes a function with the return value of the specified type.
A function type is characterized by its return value type and the number and types of its parameters.
If its return value type is T, the function is called a "function returning T".

(e) Pointer type

A pointer type may be derived from a function type, an object type, or an incomplete type, called the referenced type.
A pointer type describes an object whose value provides a reference to an entity of the referenced type.
A pointer type derived from the referenced type T is sometimes called a "pointer to T".

3.2.2 Memory

The memory model is determined by the memory space of the target device.

(1) Memory models

The following memory models are available.

Table 3-10. Memory Models

Memory Model	Maximum Values
Small modell (-ms option)	Code 64KB, Data 64KB, Total 128KB
Medium model (-mm option)	Code 1MB, Data 64KB, Total 1MB
Large model (-ml option)	Code 1MB, Data 1MB, Total 1MB

Data sections include ROM data. The above table lists maximum values when expanded functions are not used.

(2) Register banks

- The current register bank is set to "RB0" by the 78K0R C compiler startup routine. Unless it is changed, it remains set to register bank 0.
- It's set as a specified register bank at the start of the interrupt function where register bank change designation was done.

(3) Memory space

78K0R C compiler utilizes the following memory space.

Figure 3-2. Usage of Memory Space

Address		Use		Size (bytes)
00	080 - 0BFH	CALLT table		64
FF	E20 - EB3H	sreg variables, boolean variables		148
FF	EB4 - EC3H	Register variables		1616
FF	EC4 - ED3H	Compiler reserved area		16
FF	ED4 - ED7H	Segment information		4
FF	ED8 - EDFH	Runtime library arguments		8
FF	EE0 - EF7H	RB3 - RB1	Work registers[Note 1]	24
	EF8 - EFFH	RB0	Work registers	8
FF	F00 - FFFH	sfr variables		256
F0	000 - 7FFH	2nd sfr variables		Max. 2048 [Note]

Notes 1. Used when a register bank is specified.

2. Varies depending on the device used.

3.3 Extended Language Specifications

This section explains extensions unique to the 78K0R C compiler, which are not specified by theI ANSI (American National Standards Institute) standard.

The 78K0R C compiler extensions allow to generate code that makes the most effective use of the target device. These extensions are not necessarily useful in every situation, so recommended to use only those which are useful for purposes. For more information about effective use of the 78K0R C compiler extensions, see "CHAPTER 2 FUNCTIONS".

Use of the 78K0R C compiler extensions introduces microcontroller dependencies into C source programs, but compatibility on the C language level is maintained. Even if using the 78K0R C compiler extensions in C source programs, can still port the programs to other microcontrollers with a few easy-to-make modifications.

Remark In this section, RTOS" stands for the 78K0R real-time OS.

3.3.1 Macro names

78K0R C compiler defines a macro name to indicate the microcontroller name of the target device and a macro name to indicate the device name. These device names are specified by a compiling option to generate object code for the target device or by device classification in the C source code. The following examples define the macro names __K0R__and __F1166A0_.

See "(34) Predefined macro names" for more information about macro names.

```
Compiling option:
        >cc78k0r -cf1166a0 prime.c ...
```

3.3.2 Keywords

78K0R C compiler defines the following keywords to enable the extended functions. Like ANSI C keywords, these keywords cannot be used as labels or variable names.

All of these keywords are in lowercase. Any token that contains an uppercase character is not regarded as a keyword.

In the following table of keywords added by 78K0R C compiler, keywords that do not begin with "__" can be undefined by specifying the strict ANSI C conformance option (-za).

Table 3-11. Keywords Added by 78K0R C Compiler

Keyword		Purpose
Always Defined	Undefined When -za Option Is Specified	
__callt	callt	Call functions via callt table
__callf	callf	For 78K0 compatibility
__sreg	sreg	Allocate variables in saddr area
-	noauto	For 78K0 compatibility
__leaf	norec	For 78K0 compatibility
__boolean	boolean	Bit access to saddr and sfr area
-	bit	Bit access to saddr and sfr area
__interrupt	-	Hardware interrupt
__interrupt_brk	-	Software interrupt
__asm	-	ASM statements
__rtos_interrupt	-	RTOS interrupt handlers
__pascal	-	For 78K0 compatibility

Keyword		Purpose
Always Defined	Undefined When -za Option Is Specified	
__flash	-	Firmware ROM functions
__flashf	-	__flashf functions
__directmap	-	Absolute address mapping
__temp	-	For 78K0 compatibility
__near, __far	-	Memory area specification
__mxcall	-	For 78K0 compatibility

(1) Functions

The callt, __callt, __interrupt, __interrupt_brk, __rtos_interrupt, __flash, __flashf keywords are attribute qualifiers that may be added to the start of function descriptions.
The syntax is shown below.

```
attribute-qualifier ordinary-declarator function-name (parameter-type-list/identifier-list)
```

Following is an example description .

```
__callt  int  func  ( int );
```

Valid attribute qualifiers are limited to the following.
Note that callt and __callt are regarded as the same specification. However, the qualifier that begins with "__" is defined even when the -za option is specified.

- callt
- __interrupt
- __interrupt_brk
- __rtos_interrupt
- __flash
- __flashf

Caution The compiler issues a warning when it encounters the callf, __callf, noauto, __pascal, __mxcall, norec, and__leaf keywords, but otherwise ignores them.

(2) Variables

- sreg and __sreg follow the same rules as the "register" of the C language. (See " How to use the saddr area (sreg/__sreg)" for more information about the sreg keywords.)
- The bit, boolean, and __boolean type specifiers follow the same rules as the "char" and "int" type specifiers of the C language.
  However, they can be applied only to variables declared outside functions (external variables).
- The __directmap qualifier follows the same rules as the qualifiers of the C language. (See " Absolute address allocation specification (__directmap)" for details.)
- The __near and __far qualifiers follow the same rules as the type qualifiers of the C language. (See "near/far area specification" for details.)

Caution The compiler issues a warning when it encounters the __temp keyword, but otherwise ignores it.

3.3.3 #pragma directives

#pragma directives are one of the types of preprocessing directives supported by the ANSI C standard. A #pragma directive instructs the compiler to translate in a specific way, depending on the string that follows the #pragma.

When a compiler encounters a #pragma directive that it does not recognize, it ignores the directive and continues compiling. If the function of the unrecognized #pragma was to define a keyword, then an error will occur when that keyword is encountered in the C source. To avoid this, the undefined keyword should be deleted from the C source or excluded by #ifdef.

78K0R C compiler supports the following #pragma directives, which allow extended functions.

The keyword after #pragma may be specified in either uppercase or lowercase.

See "3.3.4 Using extended functions" for more information about using these directives to enable extended functions.#pragma directive list

#pragma Directive	Purpose
#pragma sfr	Use SFR names in C source files. See "How to use the sfr area (sfr)".
#pragma vect #pragma interrupt	Write interrupt service routines in C. See "Interrupt functions (#pragma vect/#pragma interrupt)".
#pragma di #pragma ei	Disable and enable interrupts in C. See "Interrupt functions (#pragma DI, #pragma EI)".
#pragma halt #pragma stop #pragma brk #pragma nop	Write CPU control instructions in C. See "CPU control instruction(#pragma HALT/STOP/BRK/NOP)".
#pragma section	Change the compiler output section name and specify the section location. See "Changing compiler output section name (#pragma section ...)".
#pragma name	Change the module name. See "Module name changing function (#pragma name)".
#pragma rot	Use the inline rotation functions. See "Rotate function (#pragma rot)".
#pragma mul	Use the inline multiplication function. See "Multiplication function (#pragma mul)".
#pragma div	Use optimized division functions. See "Division function (#pragma div)".
#pragma opc	Insert data at the current code address. See "Data insertion function (#pragma opc)".
#pragma rtos_interrupt	Write RX78K0R (real-time OS) interrupt handlers in C. See "Interrupt handler for RTOS (#pragma rtos_interrupt ...)".
#pragma rtos_task	Write RX78K0R (real-time OS) tasks in C. See "Task function for RTOS (#pragma rtos_task)".
#pragma ext_table	Specify the starting address of the flash area branch table. See "Flash area branch table and flash area allocation (#pragma ext_table)".
#pragma ext_func	Call flash area functions from boot area. See "Function of function call from boot area to flash area (#pragma ext_func)".
#pragma inline	Inline expansion of the standard library functions memcpy and memset. See "Memory manipulation function (#pragma inline)".

3.3.4 Using extended functions

The following lists the extended functions of 78K0R C compiler.

Table 3-12. Extended Function List

Extended Function	Description
callt functions (callt/__callt)	Allocated the address of a called function in the callt table area. It's possible to reduce an object code compared with usual calling instruction call..
Register variables (register)	Instructs the compiler to place a variable in a register or the saddr area, for greater execution speed. Object code is also more compact.
How to use the saddr area (sreg/__sreg)	Allocated a external variable of specified sreg or specified __sreg, and a static variable in a function in the saddr area. Variables in the saddr area can be faster execution speed than normal variables. Object code is also more compact. Variables can be allocated in the saddr area by compiler options.
Usage with saddr automatic allocation option of external variables/external static variables (-rd)	Allocated a external variable and a external static variable in the saddr area. Variables in the saddr area can be faster execution speed than normal variables. Object code is also more compact. Variables can be allocated in the saddr area by compiler options.
Usage with saddr automatic allocation option of internal static variables (-rs)	Allocated a internal static variable in the saddr area. Variables in the saddr area can be faster execution speed than normal variables. Object code is also more compact. Variables can be allocated in the saddr area by compiler options.
How to use the sfr area (sfr)	The #pragma sfr directive declares sfr names, which can use to manipulate special function registers (sfr) from C source files.
bit type variables (bit), boolean type variables (boolean/__boolean)	Generate variables having 1-bit memory area. bit and boolean/__boolean type variables allow bit access to the saddr area. boolean and __boolean type variables are functionally identical to bit type variables, and can be used in the same way.
ASM statements (#asm - #endasm/__asm)	The #asm and __asm directives allow to use assembly language statements in C source code. The statements are embedded in the assembly source code generated by the C compiler.
Kanji (2-byte character) (/* kanji */, // kanji)	C source comments can contain kanji (multibyte Japanese characters). Select the kanji encoding from Shift-JIS, EUC, or none.
Interrupt functions (#pragma vect/#pragma interrupt)	Generate the interrupt vector table, and output object code required by interrupt. This allows to write interrupt functions in C..
Interrupt function qualifier (__interrupt, __interrupt_brk)	It's possible to describe a vector table setting and an interrupt function definition in another file.
Interrupt functions (#pragma DI, #pragma EI)	Embed instructions to disable/enable interrupts in object code.
CPU control instruction(#pragma HALT/ STOP/BRK/NOP)	Embed the following instruction in object code. halt instruction stop instruction brk instruction nop instruction
Bit field declaration (Extension of type specifier)	Defining bit fields of unsigned char, signed char, signed int, unsigned short, signed short type can save memory and make object code shorter and faster execution speed.

Extended Function	Description
Bit field declaration (Allocation direction of bit field)	The -rb option changes the bit-field allocation order.
Changing compiler output section name (#pragma section ...)	Allows to change the compiler output section name and instruct the linker to locate that section independently.
Binary constant (0bxxx)	Allows specifying binary constants in C source code.
Module name changing function (#pragma name)	The module name of an object can be changed to any name in C source code.
Rotate function (#pragma rot)	Outputs the code that rotates the value of an expression to the object with direct inline expansion.
Multiplication function (#pragma mul)	Outputs the code that multiplies the value of an expression to the object with direct inline expansion. The resulting object code is smaller and faster execution speed.
Division function (#pragma div)	Output instructions using the data size of the input/output of a division instruction. The code is compatible with the 78K0 compiler. It is object code is smaller and faster execution speed than description division expressions.
BCD operation function (#pragma bcd)	Outputs the code that performs a BCD operation on the expression value in an object by direct inline expansion. BCD operation is the calculation to express 1 digit of decimal number by 4 bits of binary number.
Data insertion function (#pragma opc)	Inserts constant data into the current address. Specific data and instruction can be embedded in the code area without using the ASM statement.
Interrupt handler for RTOS (#pragma rtos_interrupt ...)	The interrupt handler for RX78K0R can be described..
Interrupt handler qualifier for RTOS (__rtos_interrupt)	The setting of the vector and the description of the interrupt handler for RX78K0R can be described in separate files.
Task function for RTOS (#pragma rtos_task)	The function names specified with #pragma rtos_task are interpreted as the tasks for RX78K0R. This allows to write efficient code of real-time OS task functions in C.
Flash area allocation method (-zf)	By compiling with the -zf option, allows programs to be allocated to the flash area, and allows those programs to be linked to object code (compiled without the -zf option) in the boot area.
Flash area branch table and flash area allocation (#pragma ext_table)	Specifies the start address of the flash area branch table by #pragma directives, allowing the startup routine and interrupt functions to be allocated in the flash area, and allowing flash area functions to be called from the boot area.
Function of function call from boot area to flash area (#pragma ext_func)	The #pragma instruction specifies the function name and ID value in the flash area called from the boot area, allowing flash area functions to be called from the boot area.
Mirror source area specification	Compiling with the -mi0/-mi1 option, instructs the compiler to generate code for a specified mirror source area.
Method of int expansion limitation of argument/ return value (-zb)	Compiling with the -zb option, to generate smaller object code and faster execution speed.
Memory manipulation function (#pragma inline)	An object file is generated by the output of the standard library functions memcpy and memset with direct inline expansion. The resulting code is faster execution speed.

Extended Function	Description
Absolute address allocation specification (__directmap)	Declare __directmap in the module in which the variable to be allocated in an absolute address is to be defined. One or more variables can be allocated to the same arbitrary address.
near/far area specification	An allocating place of the function and a variable can be designated specifically by adding the __near or __far type qualifier when a function or variable declared.
Memory model specification	An allocating place of the function and a variable can be specifying by a memory model by specifying the -ms, -mm, or -ml option when compiling.
Allocating ROM data specification	Allow to allocate ROM data in any far or near area.

callt functions (callt/__callt)

The address of the called function is allocate in the callt table area, and the function is called.

[Function]

- The callt instruction stores the address of a function to be called in an area [80H to BFH] called the callt table, so that the function can be called with a shorter code than the one used to call the function directly.
- To call a function declared by the callt (or __callt) (called the callt function), a name with ? prefixed to the function name is used. To call the function, the callt instruction is used.
- The function to be called is not different from the ordinary function.

[Effect]

- The object code can be shortened.

[Usage]

- Add the callt/__callt attribute to the function to be called as follows (described at the beginning):

```
callt   extern  type-name       function-name
__callt extern  type-name       function-name
```

[Restrictions]

- The callt functions are allocated to the area within [C0H to 0FFFFH], regardless of the memory model.
- The address of each function declared with callt/__callt will be allocated to the callt table at the time of linking object modules. For this reason, when using the callt table in an assembler source module, the routine to be created must be made "relocatable" using symbols.
- A check on the number of callt functions is made at linking time.
- When the -za option is specified, __callt is enabled and callt is disabled.
- When the -zf option is specified, callt functions cannot be defined. If a callt function is defined, an error will occur.
- The area of the callt table is 80H to BFH.
- When the callt table is used exceeding the number of callt attribute functions permitted, a compile error will occur.
- The callt table is used by specifying the -ql option. For that reason, the number of callt attributes permitted per 1 load module and the total in the linking modules is as shown below.

Option	-ql1	-ql2 to -ql3
number of callt attribute functions	32	30

- Cases where the -ql option is not used and the defaults are as shown in the table below.

callt Function	Restriction Value
Number per load module	32 max.
Total number in linked module	32 max.

[Example]

(C source)

```
============ ca1.c ============
__callt extern  int     tsub ( void );

void main ( void ) {
        int     ret_val ;
        ret_val = tsub ( );
}
```

```
 ============ ca2.c ============



__callt int     tsub ( void ) {
        int val ;
        return val ;
}
```

(Output object of compiler)

```
ca1 module
        EXTRN   ?tsub           ; Declaration
        callt   [?tsub]         ; Call

ca2 module
        PUBLIC  _tsub           ; Declaration
        PUBLIC  ?tsub           ;
@@CALT  CSEG    CALLT0          ; Allocation to segment
?tsub : DW      _tsub
@@BASE  CSEG    BASE
_tsub :                         ; Function definition
                :
                                ; Function body
                :
```

The callt attribute is given to the function tsub( ) so that it can be stored in the callt table.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified if the keyword callt/__callt is not used.
- To change functions to callt functions, observe the procedure described in the USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #define must be used. For details, see "3.3.5 C source modifications".

Register variables (register)

A variable is allocated to a register and saddr area.

[Function]

- Allocates the declared variables (including arguments of function) to the register (HL) and saddr area (_@KREG00 to _@KREG15). Saves and restores registers or saddr area during the preprocessing/ postprocessing of the module that declared a register.
- For the details of the allocation of register variables, see "3.4 Function Call Interface".
- Register variables are allocated to register HL or the saddr area (FFEB4H to FFEC3H), in the order of reference frequency. Register variables are allocated to register HL only when there is no stack frame, and allocated to the saddr area only when the -qr option is specified.

[Effect]

- Instructions to the variables allocated to the register or saddr area are generally shorter in code length than those to memory. This helps shorten object and also improves program execution speed.

[Usage]

- Declare a variable with the register storage class specifier as follows:

```
register        type-name       variable-name
```

[Restrictions]

- If register variables are not used so frequently, object code may increase (depending on the size and contents of the source).
- Register variable declarations may be used for char/int/short/long/float/double/long double and pointer data types.
- The char type uses half as much area as the int type does. The long, float, double, long double, and far pointers use twice as much area as the int type does. Between chars there are byte boundaries but in other cases, there are word boundaries.
- In the cases of int, short and near pointers, up to eight variables can be used for each function. The ninth and subsequent variables are allocated to the normal memory.
- In the case of a function without a stack frame, a maximum of 9 variables per function is usable for int, short and near pointers. The 10th and subsequent variables are allocated to the normal memory.

[Example]

<C source>

```
void func ( );

void main ( ) {
        register int    i, j ;
        i = 0 ;
        j = 1 ;
        i += j ;
        func ( );
}
```

(1) Example of register variable allocation to register HL and the saddr area

The following labels are declared in the startup routine (refer to "3.5 List of saddr Area Labels").

<Output object of compiler>

```
        EXTRN   _@KREG00        ; References the saddr area to be used
@@CODEL CSEG
_main :
        push    hl              ; Saves the contents of the register at the beginning of
                                ; the function
        movw    ax, _@KREG00    ; Saves the contents of the saddr at the beginning of
                                ; the function
        push    ax
; line  3 :     register int i, j;
; line  4 :     i = 0;
; line  5 :     j = 1;
        movw    hl, #00H        ; The following codes are output in the middle of the
                                ; function
        onew    ax
        movw    _@KREG00, ax    ; j
; line  6 :     i += j;
        addw    ax, hl
        movw    hl, ax
; line  7 :
        pop     ax              ; Restores the contents of the saddr at the end of the
                                ; function
        movw    _@KREG00, ax
        pop     hl              ; Restores the contents of the register at the end of
                                ; the function
        ret
        END
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified if the other C compiler supports register declarations.
- To change to register variables, add the register declarations for the variables to the program.

(2) From the 78K0R C compiler to another C compiler

- The C source program need not be modified if the other compiler supports register declarations.
- How many variable registers can be used and to which area they will be allocated depend on the implementations of the other C compiler.

How to use the saddr area (sreg/__sreg)

External variables that the sreg or __sreg is declared and static variables declared within functions are allocated in the saddr area.

[Function]

- The external variables and in-function static variables (called sreg variable) declared with keyword sreg or __sreg are automatically allocated to saddr area [FFE20H to FFEB3H] and with relocatability. When those variables exceed the area shown above, a compile error will occur.
- The sreg variables are treated in the same manner as the ordinary variables in the C source.
- Each bit of sreg variables of char, short, int, and long type becomes boolean type variable automatically.
- sreg variables declared without an initial value take 0 as the initial value.
- Of the sreg variables declared in the assembler source, the saddr area [FFE20H to FFF1FH] can be referred to. The area [FFEB4H to FFEDFH] are used by compiler so that care must be taken (see Figure 3-2. Usage of Memory Space).

[Effect]

- Instructions to the saddr area are generally shorter in code length than those to memory. This helps shorten object code and also improves program execution speed.

[Usage]

- Declare variables with the keywords sreg and __sreg inside a module and a function which defines the variables. Only the variable with a static storage class specifier can become a sreg variable inside a function.

```
sreg    type-name       variable-name/ sreg   static  type-name       variable-name
__sreg  type-name       variable-name/ __sreg static  type-name       variable-name
```

- Declare the following variables inside a module which refers to sreg external variables. They can be described inside a function as well.

```
extern  sreg    type-name   variable-name/ extern  __sreg  type-name   variable-name
```

[Restrictions]

- If const type is specified, or if sreg/__sreg is specified for a function, a warning message is output, and the sreg declaration is ignored.
- char type uses a half the space of other types and long/float/double/long double/far pointer types use a space twice as wide as other types.
- Between char types there are byte boundaries, but in other cases, there are word boundaries.
- When the -za option is specified, only __sreg is enabled and sreg is disabled.
- In the case of int/shortt, and near pointer and pointer, a maximum of 74 variables per load module is usable (when saddr area [FFE20H to FFEB3H] is used).
  Note that the number of usable variables decreases when bit and boolean type variables, boolean type variables are used.

[Example]

<C source>

```
extern sreg     int     hsmm0 ;
extern sreg     int     hsmm1 ;
extern sreg     int     *hsptr ;

void main ( ) {
        hsmm0 -= hsmm1 ;
}
```

The following example shows a definition code for sreg variable that the user creates. If extern declaration is ot made in the C source, the 78K0R C compiler outputs the following codes. In this case, the ORG quasi-directive will not be output.

```
        PUBLIC  _hsmm0  ; Declaration
        PUBLIC  _hsmm1  ;
        PUBLIC  _hsptr  ;

@@DATS  DSEG    SADDRP  ; Allocation to segment
        ORG     0FE20H  ;
_hsmm0 :        DS      ( 2 )   ;
_hsmm1 :        DS      ( 2 )   ;
_hsptr :        DS      ( 2 )   ;
```

The following codes are output in the function.

```
movw    ax, _hsmm0
subw    ax, _hsmm1
movw    _hsmm0, ax
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed if the other compiler does not use the keyword sreg/__sreg.
  To change to sreg variable, modifications are made according to the method shown above.

(2) From the 78K0R C compiler to another C compiler

- Modifications are made by #define. For the details, see "3.3.5 C source modifications". Thereby, sreg variables are handled as ordinary variables.

Usage with saddr automatic allocation option of external variables/external static variables (-rd)

The -rd option to automatically allocate external variables and external static variables in the saddr area.

[Function]

- External variables/external static variables (except const type) are automatically allocated to the saddr area regardless of whether sreg declaration is made or not.
- Depending on the value of *n* and the specification of m, the external static variables and external static variables to allocate can be specified as follows.

Specification of *n*,m	Variables Allocated to saddr Area
n	(1) When *n* = 1 Variables of char and unsigned char types (2) When *n* = 2 Variables for when *n* = 1, plus variables of short, unsigned short, int, unsigned int, enum, and near pointer type When n = 4 Variables for when n = 2, plus variables of long, unsigned long, float, double, and long double, far pointer type
m	Structures, unions, and arrays
When omitted	All variables

- Variables declared with the keyword sreg are allocated to the saddr area, regardless of the above specification.
- The above rule also applies to variables referenced by extern declaration, and processing is performed as if these variables were allocated to the saddr area.
- The variables allocated to the saddr area by this option are treated in the same manner as the sreg variable. The functions and restrictions of these variables are as described in [How to use the saddr area (sreg/__sreg)].

[Method of specification]

- Specify the -rd[*n*][m] (*n* = 1, 2, or 4) option.

[Restrictions]

- In the -rd[*n*][m] option, modules specifying different n, m value cannot be linked each other.

Usage with saddr automatic allocation option of internal static variables (-rs)

The -rs option to automatically allocate internal static variables in the saddr area.

[Function]

- Automatically allocates internal static variables (except const type) to saddr area regardless of with/ without sreg declaration.
- Depending on the value of *n* and the specification of m, the internal static variables to allocate can be specified as follows.

Specification of *n*, m	Variables Allocated to saddr Area
n	(1) When *n* = 1: Variables of char and unsigned char types (2) When *n* = 2: Variables for when *n* = 1, plus variables of short, unsigned short, int, unsigned int, enum, and near pointer type (3) When *n* = 4: Variables for when *n* = 2, plus variables of long, unsigned long, float, double, and long double, far pointer type
m	Structures, unions, and arrays
When omitted	All variables (including structures, unions, and arrays in this case only)

- Variables declared with the keyword sreg are allocated to the saddr area regardless of the above specification.
- The variables allocated to the saddr area by this option are handled in the same manner as the sreg variable. The functions and restrictions for these variables are as described in [How to use the saddr area (sreg/__sreg)].

[Method of specification]

- Specify the -rs[*n*][m] (*n* = 1, 2, or 4) option.

Remark In the -rs[*n*][m] option, modules specifying different n, m value can also be linked each other.

How to use the sfr area (sfr)

The #pragma sfr directive declares sfr names, which can use to manipulate special function registers (sfr) from C source files.

[Function]

- The sfr area refers to a group of special function registers such as mode registers and control registers for the various peripherals of the 78K0R microcontrollers.
- By declaring use of sfr names, manipulations on the sfr area can be described at the C source level.
- sfr variables are external variables without initial value (undefined).
- A write check will be performed on read-only sfr variables.
- A read check will be performed on write-only sfr variables.
- Assignment of an illegal data to an sfr variable will result in a compile error.
- The sfr names that can be used are those allocated to an area consisting of addresses [FFF00H to FFFFFH, and F0000H to F07FFH[Note]].

Note Varies depending on the device used.

[Effect]

- Manipulations to the sfr area can be described in the C source level.
- Instructions to the sfr area are shorter in code length than those to memory. This helps shorten object code and also improves program execution speed.

[Usage]

- Declare the use of an sfr name in the C source with the #pragma preprocessor directive, as follows (The keyword sfr can be described in uppercase or lowercase letters.):

```
#pragma sfr
```

The #pragma sfr directive must be described at the beginning of the C source line.
If #pragma PC (processor type) is specified, however, describe #pragma sfr after that.
The following statement and directives may precede the #pragma sfr directive:

- Comment
- Preprocessor directives which do not define nor refer to a variable or function

- In the C source program, describe an sfr name that the device has as is (without change). In this case, the sfr need not be declared.

[Restrictions]

- All sfr names must be described in uppercase letters. Lowercase letters are treated as ordinary variables.

[Example]

<C source>

```
#ifdef  __K0R__
#pragma sfr
#endif

void main ( void ) {
        PL0 -= ADCR ;
        /* ADCR = 10 ;  ==> error */
}
```

Codes that relate to declarations are not output and the following codes are output in the middle of the function.

```
mov     a, PL0
sub     a, ADCR
mov     PL0, a
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Those portions of the C source program not dependent on the device or compiler need not be modified.

(2) From the 78K0R C compiler to another C compiler

- Delete the "#pragma sfr" statement or sort by "#ifdef" and add the declaration of the variable that was formerly a sfr variable.
  The following shows an example.

```
#ifdef  __K0R__
#pragma sfr
#else
/*Declaration of variables*/
unsigned char   P0 ;
#endif

void main ( void ) {
        P0 = 0 ;
}
```

- In case of a device which has the sfr or its alternative functions, a dedicated library must be created to access that area.

bit type variables (bit), boolean type variables (boolean/__boolean)

The bit, boolean, and __boolean type specifiers generate variables having 1-bit of memory area.

[Function]

- A bit or boolean type variable is handled as 1-bit data and allocated to saddr area.
- This variable can be handled the same as an external variable that has no initial value (or has an unknown value).
- To this variable, the C compiler outputs the following bit manipulation instructions:

```
MOV1 , AND1 , OR1 , XOR1 , SET1 , CLR1 , NOT1 , BT , BF
```

[Effect]

- Programming at the assembler source level can be performed in C, and the saddr and sfr area can be accessed in bit units.

[Usage]

- Declare a bit or boolean type inside a module in which the bit or boolean type variable is to be used, as follows:
- __boolean can also be described instead of bit.

```
bit             variable-name
boolean         variable-name
__boolean       variable-name
```

- Declare a bit or boolean type inside a module in which the bit or boolean type variable is to be used, as follows:

```
extern  bit             variable-name
extern  boolean         variable-name
extern  __boolean       variable-name
```

- char, int, short, and long type sreg variables (except the elements of arrays and members of structures) and 8-bit sfr variables can be automatically used as bit type variables.

```
variable-name.n (where n = 0 to 31)
```

[Restrictions]

- An operation on 2 bit or boolean type variables is performed by using the CY (Carry) flag. For this reason, the contents of the carry flag between statements are not guaranteed.
- Arrays cannot be defined or referenced.
- A bit or boolean type variable cannot be used as a member of a structure or union.
- This type of variable cannot be used as the argument type of a function.
- A bit type variable cannot be used as a type of automatic variable.
- With bit type variables only, up to 1184 variables can be used per load module (when saddr area [FFE20H to FFEB3H] is used) (normal model).
- The variable cannot be declared with an initial value.
- If the variable is described along with const declaration, the const declaration is ignored.

- Only operations using 0 and 1 can be performed by the operators and constants shown in the table below.

Classification	Operator
Assignment	=
Bitwise AND	&, &=
Bitwise OR	|, |=
Bitwise XOR	^, ^=
Logical AND	&&
Logical OR	||
Equal	==
Not Equal	!=

- *, & (pointer reference, address reference), and sizeof operations cannot be performed.
- When the -za option is specified, only __boolean is enabled.
- In the case that sreg variables are used or if -rd, -rs (saddr automatic allocation option) options are specified, the number of usable bit type variables is decreased.

[Example]

<C source>

```
#define ON      1
#define OFF     0

extern  bit     data1 ;
extern  bit     data2 ;

void main ( void ) {
        data1 = ON ;
        data2 = OFF ;
        while ( data1 )  {
                data1 = data2;
                testb ( );
        }

        if ( data1 && data2 ) {
                chgb ( );
        }
}
```

This example is for cases when the user has generated a definition code for a bit type variable. If an extern declaration has not been attached, the compiler outputs the following code. The ORG quasi-directive is not output in this case.

```
PUBLIC  _data1          ; Declaration
PUBLIC  _data2

@@BITS  BSEG            ; Allocation to segment
        ORG     0FE20H
_data1  DBIT
_data2  DBIT
```

The following codes are output in a function

```
set1    _data1                  (Initialized)
clr1    _data2                  (Initialized)
bf      data1, $?L0004          (Judgment)
mov1    CY, _data2              (Assignment)
mov1    _data1, CY              (Assignment)
bf      _data1, $?L0005         (Logical AND expression)
bf      _data2, $?L0005         (Logical AND expression)
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified if the keyword bit, boolean, or __boolean is not used.
- To change variables to bit or boolean type variables, modify the program according to the procedure described in USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #define must be used. For details, see "3.3.5 C source modifications" (As a result of this, the bit or boolean type variables are handled as ordinary variables.).

ASM statements (#asm - #endasm/__asm)

The #asm and __asm directives allow to use assembly language statements in C source code. The statements are embedded in the assembly source code generated by the C compiler.

[Function]

(1) #asm - #endasm

- The assembler source program described by the user can be embedded in an assembler source file to be output by the 78K0R C compiler by using the preprocessor directives #asm and #endasm.
- #asm and #endasm lines will not be output.

(2) __asm

- An assembly instruction is output by describing an assembly code to a character string literal and is inserted in an assembler source.

[Effect]

- To manipulate the global variables of the C source in the assembler source
- To implement functions that cannot be described in the C source
- To hand-optimize the assembler source output by the C compiler and embed it in the C source (to obtain efficient object)

[Usage]

(1) #asm #endasm

- Indicate the start of the assembler source with the #asm directive and the end of the assembler source with the #endasm directive. Describe the assembler source between #asm and #endasm.

```
#asm
  :     /* Assembler source */
#endasm
```

(2) __asm

- The ASM statement is described in the following format in the C source:

```
__asm (string-literal);
```

- The description method of character string literal conforms to ANSI, and a line can be continued by using an escape character string (\n: line feed, \t: tab) or \, or character strings can be linked.

[Restrictions]

- Nesting of #asm directives is not allowed.
  If ASM statements are used, no object module file will be created. Instead, an assembler source file will be created.
  [Output assemble file] by [Compile Options] tab of Property panel, set it as "Yes." (See the "CubeSuite 78K0R build" for a setting method.)
- Only lowercase letters can be described for __asm. If __asm is described with uppercase and lowercase characters mixed, it is regarded as a user function.
- When the -za option is specified, only __asm is enabled.
- #asm - #endasm and __asm block can only be described inside a function of the C source. Therefore, the assembler source is output to CSEG of segment name @@CODE, or @@CODEL.

[Example]

(1) #asm - #endasm

<C source>

```
void main ( void ) {
#asm
        callt [init]
#endasm
}
```

<Output object of compiler>

```
@@CODEL CSEG
_main :
        callt [init]
        ret
        END
```

In the above example, statements between #asm and #endasm will be output as an assembler source program to the assembler source file.

(2) __asm

<C source>

```
int     a, b ;

void main ( void ) {
        __asm ( "\tmovw ax, !_a \t ; ax <- a" );
        __asm ( "\tmovw !_b, ax \t ; b <- ax" );
}
```

<Output object of compiler>

```
@@CODEL CSEG
_main :
        movw    ax, !_a         ; ax <- a
        movw    !_b, ax         ; b <- ax
        ret
        END
```

[Compatibility]

- With the C compiler which supports #asm, modify the program according to the format specified by the C compiler.
- If the target device is different, modify the assembler source part of the program.

Kanji (2-byte character) (/* kanji */, // kanji)

C source comments can contain kanji (multibyte Japanese characters).

[Function]

- Kanji code can be described in comments in C source files.
- Kanji code in comments is handled as a part of comments, so the code is not subject to compilation.
- The kanji code to be used in comments can be specified by using an option or the environment variable.
  If no option is specified, the code set in the environment variable LANG78K is set as the kanji code.
- If the kanji code is specified by both the option and environment variable LANG78K, specification by the option takes precedence.
- If "SJIS" is set in the environment variable LANG78K, the type of kanji in comments is Interpreted as shift JIS code.
- If "EUC" is set in the environment variable LANG78K, the compiler interprets this as meaning that the type of kanji in comments is EUC code.
- If "NONE" is set in the environment variable LANG78K, the compiler interprets this as meaning that comments do not contain kanji codes.
- SJIS code is specified by default.

[Effect]

- The use of kanji code allows Japanese programmers to describe easier-to-understand comments, which makes C source management easier.

[Usage]

- Set the kanji code by using a compiler option or environment variable. (Setting is not needed if the default setting is used.)

(1) Setting by compiler option

Set any of the options listed in the following table.

Option	Explaination
-zs	SJIS (shift JIS code)
-ze	EUC (EUC code)
-zn	NONE (kanji code not used)

(2) Setting by environment variable LANG78K

- Set "SJIS", "EUC" or "NONE". (Also describe it in files such as autoexec.bat, if necessary.)
- Specification of SJIS, EUC or NONE is not case-sensitive.
- Describe kanji characters in comments in C source files, in accordance with the one specified in LANG78K.

```
SET     LANG78K = SJIS  (shift JIS code)
SET     LANG78K = EUC   (EUC code)
SET     LANG78K = NONE  (kanji code not used)
```

[Restrictions]

- Only shift JIS code and EUC code can be described in comments. Only the characters of 0x7F or lower ASCII codes can be described for places other than comments. Neither full-size characters nor half-size katakana (including half-size punctuation marks) can be described for any place other than comments.
  If any of these characters is described, the expected code may not be output.

[Example]

<C source>

```
// main function
void main ( void ) {
        /* Comment */
}
```

Kanji type information is output to the assembler source.

<Output object of compiler>

```
$KANJICODE SJIS
```

When the C source contents are output to the assembler source, kanji characters in the comment are also output.

```
; line          1 : // main function
; line          2 : void main ( void ) {
@@CODEL CSEG
_main :
; line          3 :             /* Comment */
; line          4 : }
```

[Description]

- Kanji code can be described only in comments in C source files.
- When using the format "// comment", specify compiler option -zp.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- If there is kanji in the the area that comment cannot be described (the area other than "/* ... */", or "// new-line"), the source files must be modified.
- If the kanji code differs from the one specified in the CC78K0R, the kanji code must first be converted.

(2) From the 78K0R C compiler to another C compiler

- The C source need not be modified for a C compiler that supports kanji characters to be described in comments.
- Kanji characters in the C source must be deleted if the C compiler does not support kanji characters to be described in comments.

Interrupt functions (#pragma vect/#pragma interrupt)

Generate the interrupt vector table, and output object code required by interrupt.

[Function]

- The address of a described function name is registered to an interrupt vector table corresponding to a specified interrupt request name.
- An interrupt function outputs a code to save or restore the following data (except that used in the ASM statement) to or from the stack at the beginning and end of the function (after the code if a register bank is specified):
  - Registers
  - saddr area for register variables
  - saddr area for work
  - saddr area for run time library
  - saddr area for storing segment information
  - ES and CS registers

  Note, however, that depending on the specification or status of the interrupt function, saving/restoring is performed differently, as follows:
  - If no change is specified, codes that change the register bank or saves/restores register contents, and that saves/restores the contents of the saddr area are not output regardless of whether to use the codes or not.
  - If a register bank is specified, a code to select the specified register bank is output at the beginning of the interrupt function, therefore, the contents of the registers are not saved or restored.
  - If no change is not specified and if a function is called in the interrupt function, however, the entire register area is saved or restored, regardless of whether use of registers is specified or not.

- If the -qr option is not specified for compilation, the saddr area for register variables and the saddr area for work are not used; so the saving/restoring code is not output.
  If the size of the saving code is smaller than that of the restoring code, the restoring code is output. The table below summarizes the above and lists the saving/restoring areas.

Save/Restore Area	NO BANK	Function Called				Function Not Called			
		Without -qr		With -qr		Without -qr		With -qr	
		Stack	RBn	Stack	RBn	Stack	RBn	Stack	RBn
Register used	NG	NG	NG	NG	NG	OK	NG	OK	NG
All registers	NG	OK	NG	OK	NG	NG	NG	NG	NG
saddr area for runtime library used, ES, CS register, saddr area for storing segment information	NG	NG	NG	NG	NG	OK	OK	OK	OK
saddr area for all runtime libraries, ES, CS register, saddr area for storing segment information	NG	OK	OK	OK	OK	NG	NG	NG	NG
saddr area for register variable used	NG	NG	NG	OK	OK	NG	NG	OK	OK
All saddr area for work	NG	NG	NG	OK	OK	NG	NG	NG	NG

Stack : Use of stack is specified
RBn : Register bank is specified
OK : Saved
NG : Not saved

[Effect]

- Interrupt functions can be described at the C source level.
- Because the register bank can be changed, codes that save the registers are not output; therefore, object codes can be shortened and program execution speed can be improved.
- You do not have to be aware of the addresses of the vector table to recognize an interrupt request name.

[Usage]

- Specify an interrupt request name, a function name, stack switching, registers used by the compiler, and whether the saddr area is saved/restored, with the #pragma directive. Describe the #pragma directive at the beginning of the C source. The #pragma directive is described at the start of the C source (for the interrupt request names, see the user's manual of the target device used). For the software interrupt BRK, describe BRK_I.
- The following items can be described before this #pragma directive:
  - Comments
  - Preprocessor directive which does neither define nor refer to a variable or a function

```
#pragmaΔvect(or interrupt)Δinterrupt-request-nameΔfunction-nameΔ

              [Stack-change-specification]┌Δ  Stack-usage-specification ┐
                                          │    No-change-specification  │
                                          └    Register-bank-specification┘
```

- *Interrupt request name*
  Described in uppercase letters.
  See the user's manual of the target device used (example: NMI, INTP0, etc.).
  For the software interrupt BRK, describe BRK_I.
- *Function name*
  Name of the function that describes interrupt processing
- *Stack change specification*
  SP = array name [+ offset location] (example: SP = buff + 10)
  Define the array by unsigned short (example: unsigned short buff [5];).
  Specify for the offset location an even value of the buff size or lower. (Example: In the case of unsigned short buff[5], the buff size is 10 bytes, so an even value of 10 or lower should be specified.)
- *Stack use specification*
  STACK (default)
- *No change specification*
  NOBANK
- *Register bank specification*
  RB0/RB1/RB2/RB3
- Δ
  Space

Caution Since the 78K0R C compiler startup routine is initialized to register bank 0, be sure to specify register banks 1 to 3.

[Restrictions]

- When the -zf option is not specified, interrupt functions are allocated to the area between C0H and 0FFFFH, regardless of the memory model.
  When the -zf option is specified, interrupt functions are allocated in accordance with the memory model. In addition, specification of allocation area by specifying __near or __far is also enabled.
- Arrays in an area other than the near area cannot be specified for stack change. If specified, an error will occur.
- A value other than an even value cannot be specified for the offset location. If specified, an error will occur.
- Unlike other microcontrollers, the unsigned short type array is reserved for changing the stack pointer.
- An interrupt request name must be described in uppercase letters.
- A duplication check on interrupt request names will be made within only 1 module.
- The contents of a register may be changed if the following three conditions are satisfied, but the compiler cannot check this.
  If it is specified to change the register bank, set the register banks so that they do not overlap. If register banks overlap, control their interrupts so that they do not overlap.
  When NOBANK (no change specification) is specified, the registers are not saved. Therefore, control the registers so that their contents are not lost.
  - If two or more interrupts occur
  - If two or more interrupts that use the same BANK are included in the interrupt that has occurred
  - If NOBANK or a register bank is specified in the description #pragma interrupt -.
- As the interrupt function, callt/__callt/__rtos_interrupt/__flash/__flashf cannot be specified.
  __far can be specified only when the -zf option is specified.
- An interrupt function is specified with void type (example: void func (void);) because it cannot have an argument nor return value.
- Even if an ASM statement exists in the interrupt function, codes saving all the registers and variable areas are not output. If an area reserved for the compiler is used in the ASM statement in the interrupt function, therefore, or if a function is called in the ASM statement, the user must save the registers and variable areas.
- If leafwork 1 to 16 is specified, a warning is output and the specification is ignored.
- When stack change is specified, the stack pointer is changed to the location where offset is added to the array name symbol. The area of the array name is not secured by the #pragma directive. It needs to be defined separately as global unsigned short type array.
- The code that changes the stack pointer is generated at the start of a function, and the code that sets the stack pointer back is generated at the end of a function.
- When keywords sreg/__sreg are added to the array for stack change, it is regarded that two or more variables with the different attributes and the same name are defined, and a compile error will occur. It is possible to allocate an array in saddr area by the -rd option, but code and speed efficiency will not be improved because the array is used as a stack. It is recommended to use the saddr area for purposes other than a stack.
- The stack change cannot be specified simultaneously with the no change. If specified so, an error will occur.
- The stack change must be described before the stack use/register bank specification. If the stack change is described after the stack use/register bank specification, an error will occur.
- If a function specifying no change, register bank, or stack change as the saving destination in #pragma vect/ #pragma interrupt specification is not defined in the same module, a warning message is output and the stack change is ignored. In this case, the default stack is used.

[Example]

(1) When register bank is specified

<C source>

```
#pragma interrupt INTP0 inter rb1

void inter ( void ) {
        /* Interrupt processing to INTP0 pin input*/
}
```

<Output object of compiler>

```
@@VECT08        CSEG    AT      0008H ; INTP0
_@vect08 :
        DW      _inter
@@BASE          CSEG    BASE
_inter :

        ; Switching code for the register bank
        ; Saving code of the saddr area for use by the compiler
        ; Saves ES and CS registers
        ; Interrupt processing to INTP0 pin input (function body)
        ; Restores ES and CS registers
        ; Restoring code of the saddr area used by the compiler
        reti
```

(2) When stack change and register bank are specified

<C source>

```
#pragma interrupt INTP0 inter sp = buff + 10 rb2

unsigned short  buff[5] ;
void func ( void );

void inter ( void ) {
        func ( );
}
```

<Output object of compiler>

```
@@BASE      CSEG    BASE
_inter :
        sel     RB2             ; Changes register bank
        movw    ax, sp          ; Changes stack pointer
        movw    sp, #_buff + 10 ;      :
        push    ax              ;      :
        movw    c, #0CH         ; Saves saddr used by the compiler
        dec     c               ;      :
        dec     c               ;      :
        movw    ax, _@SEGAX[c]  ;      :
        push    ax              ;      :
        bnz     $$ - 6          ;      :
        mov     a, ES           ; Saves ES and CS registers
        mov     x, a            ;      :
        mov     a, CS           ;      :
        push    ax              ;      :
        call    !!_func
        pop     ax              ; Restores ES and CS registers
        mov     CS, a           ;      :
        mov     a, x            ;      :
        mov     CS, a           ;      :
        movw    de, #_@SEGAX    ; Restores saddr used by the compiler
        mov     c, #06H         ;      :
        pop     ax              ;      :
        movw    [de], ax        ;      :
        incw    de              ;      :
        incw    de              ;      :
        dec     c               ;      :
        bnz     $$ - 5          ;      :
        pop     ax              ; Returns the stack pointer to its original position
        movw    sp, ax          ;      :
        reti

@@VECT08    CSEG    AT      0008H
_@vect08 :
            DW      _inter
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified if interrupt functions are not used at all.
- To change an ordinary function to an interrupt function, modify the program according to the procedure described in USAGE above.

(2) From the 78K0R C compiler to another C compiler

- An interrupt function can be used as an ordinary function by deleting its specification with the #pragma vect, #pragma interrupt directive.
- When an ordinary function is to be used as an interrupt function, change the program according to the specifications of each compiler.

Interrupt function qualifier (__interrupt, __interrupt_brk)

It's possible to describe a vector table setting and an interrupt function definition in another file.

[Function]

- A function declared with the __interrupt qualifier is regarded as a hardware interrupt function, and execution is returned by the return RETI instruction for non-maskable/maskable interrupt function.
- By declaring a function with the __interrupt_brk qualifier, the function is regarded as a software interrupt function, and execution is returned by the return instruction RETB for software interrupt function.
- A function declared with this qualifier is regarded as (non-maskable/maskable/software) interrupt function, and saves or restores the registers and variable areas (1) and (6) below, which are used as the work area of the compiler, to or from the stack.
  If a function call is described in this function, however, all the variable areas are saved to the stack.

(1) Registers

(2) saddr area for register variables

(3) saddr area for work

(4) saddr area for run time library

(5) saddr area for storing segment information

(6) ES and CS registers

Remark If the -qr option is not specified (default) at compile time, save/restore codes are not output because areas (2) and (3) are not used.

[Effect]

- By declaring a function with this qualifier, the setting of a vector table and interrupt function definition can be described in separate files.

[Usage]

- Describe either __interrupt or __interrupt_brk as the qualifier of an interrupt function.

(1) For non-maskable/maskable interrupt function

```
__interrupt     void func ( ) {processing}
```

(2) For software interrupt function>

```
__interrupt_brk void func ( ) {processing}
```

[Restrictions]

- When the -zf option is specified, the interrupt functions are allocated to the area within [C0H to 0FFFFH], regardless of the memory model.
  When the -zf option is specified, interrupt functions are allocated in accordance with the memory model. In addition, specification of allocation area by specifying __near or __far is also enabled.
- The interrupt function cannot specify callt/__callt/__rtos_interrupt/__flash/__flashf.

[Cautions]

- The vector address is not set by merely declaring this qualifier. The vector address must be separately set by using the #pragma vect/interrupt directive or assembler description.
- The saddr area and registers are saved to the stack.
- Even if the vector address is set or the saving destination is changed by #pragma vect (or interrupt) ..., the change in the saving destination is ignored if there is no function definition in the same file, and the default stack is assumed.
- To define an interrupt function in the same file as the #pragma vect (or interrupt) ... specification, the function name specified by #pragma vect (or interrupt) ... is judged as the interrupt function, even if this qualifier is not described.
  For details of #pragma vect/interrupt, see USAGE of "Interrupt functions (#pragma vect/#pragma interrupt)".

[Example]

- Declare or define interrupt functions in the following format. The code to set the vector address is generated by #pragma interrupt.

```
#pragma interrupt       INTP0   inter   RB1     /*The interrupt request name of*/
#pragma interrupt       BRK_I   inter_b RB2     /*The software interrupt is "BRK_I"*/

__interrupt     void inter ( );                 /*Prototype declaration*/
__interrupt_brk void inter_b ( );               /*Prototype declaration*/
__interrupt     void inter ( ) {processing};    /*Function body*/
__interrupt_brk void inter_b ( ) {processing};  /*Function body*/
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified unless interrupt functions are supported.
- Modify the interrupt functions, if necessary, according to the procedure described in USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #define must be used to allow the interrupt qualifiers to be handled as ordinary functions.
- To use the interrupt qualifiers as interrupt functions, modify the program according to the specifications of each compiler.

Interrupt functions (#pragma DI, #pragma EI)

Embed instructions to disable/enable interrupts in object code.

[Function]

- Codes DI and EI are output to the object and an object file is created.
- If the #pragma directive is missing, DI( ) and EI( ) are regarded as ordinary functions.
- If "DI( );" is described at the beginning in a function (except the declaration of an automatic variable, comment, and preprocessor directive), the DI code is output before the preprocessing of the function (immediately after the label of the function name).
- To output the code of DI after the preprocessing of the function, open a new block before describing "DI( );" (delimit this block with "{").
- If "EI( );" is described at the end of a function (except comments and preprocessor directive), the EI code is output after the post-processing of the function (immediately before the code RET).
- To output the EI code before the post-processing of a function, close a new block after describing "EI( );" (delimit this block with "}").

[Effect]

- A function disabling interrupts can be created.

[Usage]

- Describe the #pragma DI and #pragma EI directives at the beginning of the C source.
  However, the following statement and directives may precede the #pragma DI and #pragma EI directives:
  - Comment
  - Other #pragma directives
  - Preprocessor directive which does neither define nor refer to a variable or function
- Describe DI( ); or EI( ); in the source in the same manner as function call.
- DI and EI can be described in either uppercase or lowercase letters after #pragma.

[Restrictions]

- When using these interrupt functions, DI and EI cannot be used as function names.
- DI and EI must be described in uppercase letters. If described in lowercase letters, they will be handled as ordinary functions.

[Example]

```
#ifdef  __K0R__
#pragma DI
#pragma EI
#endif
```

<C source>

```
#pragma DI
#pragma EI

void main ( void ) {
        DI ( );
        ;Function body
        EI ( );
}
```

<Output object of compiler>

```
_main :
        di
        ; Preprocessing
        ; Function body
        ; Postprocessing
        ei
        ret
```

(1) To output DI and EI after and before preprocessing/post-processing

<C source>

```
#pragma DI
#pragma EI

void main ( void ) {
        {
                DI ( );
                ; Function body
                EI ( );
        }
}
```

<Output object of compiler>

```
_main :
        ; Preprocessing
        di
        ; Function body
        ei
        ; Postprocessing
        ret
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified if interrupt functions are not used at all.
- To change an ordinary function to an interrupt function, modify the program according to the procedure described in USAGE above.

(2) From the 78K0R C compiler to another C compiler

- Delete the #pragma DI and #pragma EI directives or invalidate these directives by separating them with #ifdef and DI and EI can be used as ordinary function names (example: #ifdef__K0R__ ... #endif).
- When an ordinary function is to be used as an interrupt function, modify the program according to the specifications of each compiler.

CPU control instruction(#pragma HALT/STOP/BRK/NOP)

The #pragma HALT/STOP/BRK/NOP directives declare functions that embed CPU control instructions.

[Function]

- The following codes are output to the object to create an object file:
- Instruction for HALT operation (HALT)
- Instruction for STOP operation (STOP)
- BRK instruction
- NOP instruction

[Effect]

- The standby function of a microcontroller can be used with a C program.
- A software interrupt can be generated.
- The clock can be advanced without the CPU operating.

[Usage]

- Describe the #pragma HALT, #pragma STOP, #pragma NOP, and #pragma BRK instructions at the beginning of the C source.
- The following items can be described before the #pragma directive:
  - Comment
  - Other #pragma directive
  - Preprocessor directive which does neither define nor refer to a variable or function
- The keywords following #pragma can be described in either uppercase or lowercase letters.
- Describe as follows in uppercase letters in the C source in the same format as function call:

```
HALT ( );
STOP ( );
BRK ( );
NOP ( );
```

[Restrictions]

- When this feature is used, HALT, STOP, BRK, and NOP cannot be used as function names.
- Describe HALT, STOP, BRK, and NOP in uppercase letters. If they are described in lowercase letters, they are handled as ordinary functions.

[Example]

<C source>

```
#pragma HALT
#pragma STOP
#pragma BRK
#pragma NOP

void main ( void ) {
        HALT ( );
        STOP ( );
        BRK ( );
        NOP ( );
}
```

<Output object of compiler>

```
@@CODEL CSEG
_main :
        halt
        stop
        brk
        nop
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The C source program need not be modified if the CPU control instructions are not used.
- Modify the program according to the procedure described in USAGE above when the CPU control instructions are used.

(2) From the 78K0R C compiler to another C compiler

- If "#pragma HALT", "#pragma STOP", "#pragma BRK", and "#pragma NOP" statements are delimited by means of deletion or with #ifdef, HALT, STOP, BRK, and NOP can be used as function names.
- To use these instructions as the CPU control instructions, modify the program according to the specifications of each compiler.

Bit field declaration (Extension of type specifier)

It's possible declare bit-fields of type unsigned char, signed char, unsigned int, signed int, unsigned short, and signed short.

[Function]

- The bit field of unsigned char, signed char type is not allocated straddling over a byte boundary.
- The bit field of unsigned int, signed int, unsigned short, signed short type is not allocated straddling over a word boundary, but can be allocated straddling over a word boundary when the -rc option is specified.
- The bit fields that the types are same size are allocated in the same byte units (or word units).
  If the types are different size, the bit fields are allocated in different byte units (or word units).
- unsigned short, signed short type is handled similarly with unsigned int, signed int type respectively.

[Effect]

- The memory can be saved, the object code can be shortened, and the execution speed can be improved.

[Usage]

- As a bit field type specifier, unsigned char, signed char, signed int, unsigned short, signed short type can be specified in addition to unsigned int type.
  Declare as follows.

```
struct  tag-name {
        unsigned char   field-name : bit-width ;
        unsigned char   field-name : bit-width ;
                :
        unsigned int    field-name : bit-width ;
};
```

[Example]

```
struct  tagname {
        unsigned char   A : 1 ;
        unsigned char   B : 1 ;
                :
        unsigned int    C : 2 ;
        unsigned int    D : 1 ;
                :
};
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The source program need not be modified.
- Change the type specifier to use unsigned char, signed char, unsigned short, signed short as the type specifier.

(2) From the 78K0R C compiler to another C compiler

- The source program need not be modified if unsigned char, signed char, signed int, unsigned short and signed short is not used as a type specifier.
- Change into unsigned int, if unsigned char, signed char, signed int, unsigned short and signed short is used as a type specifier.

Bit field declaration (Allocation direction of bit field)

The -rb option changes the bit-field allocation order.

[Function]

- The direction in which a bit field is to be allocated is changed and the bit field is allocated from the MSB side when the -rb option is specified.
- If the -rb option is not specified, the bit field is allocated from the LSB side.

[Usage]

- Specify the -rb option at compile time to allocate the bit field from the MSB side.
- Do not specify the option to allocate the bit field from the LSB side.

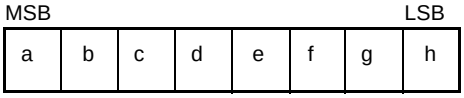
[Example]

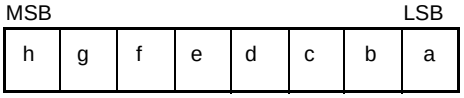
(1) Bit field declaration 1

```
struct  t {
        unsigned char   a : 1 ;
        unsigned char   b : 1 ;
        unsigned char   c : 1 ;
        unsigned char   d : 1 ;
        unsigned char   e : 1 ;
        unsigned char   f : 1 ;
        unsigned char   g : 1 ;
        unsigned char   h : 1 ;
};
```

Because a through h are 8 bits or less, they are allocated in 1-byte units.

Bit allocation from MSB with the -rb option specified	Bit allocation from LSB without the -rb option specified

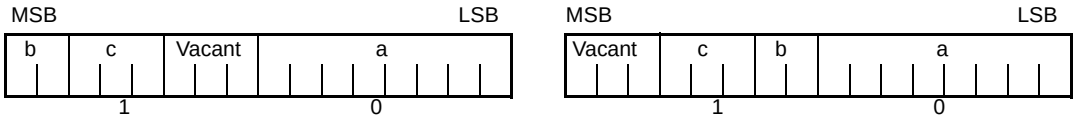




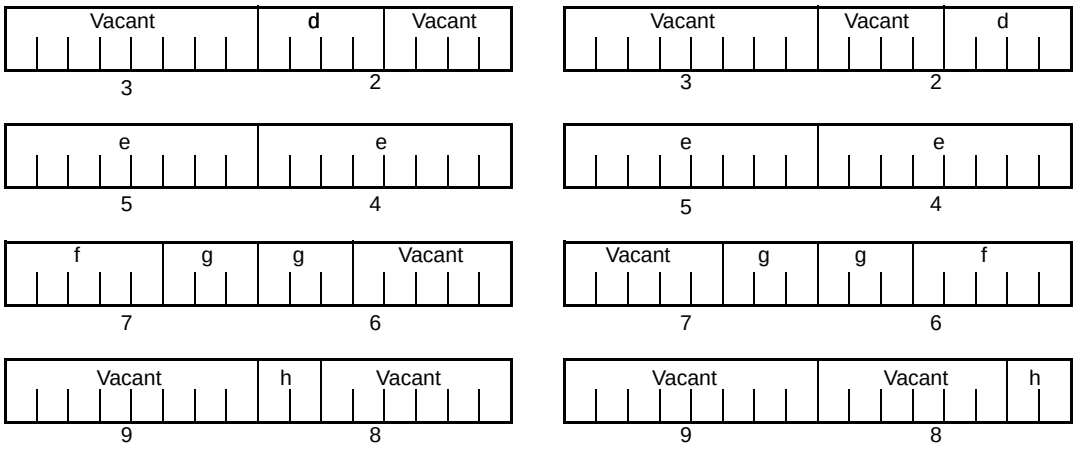
(2) Bit field declaration 2

```
struct  t {
        char            a ;
        unsigned char   b : 2 ;
        unsigned char   c : 3 ;
        unsigned char   d : 4 ;
        int             e ;
        unsigned int    f : 5 ;
        unsigned int    g : 6 ;
        unsigned char   h : 2 ;
        unsigned int    i : 2 ;
};
```

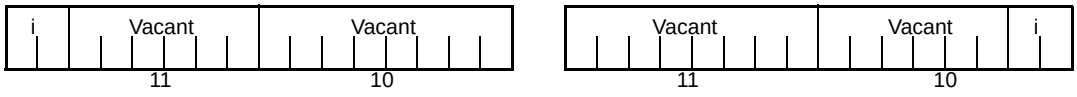
Bit field allocated from the MSB side when the -rb option is specified	Bit field allocated from the LSB side when the -rb option is not specified



Member a of char type is allocated to the first byte unit. Members b and c are allocated to subsequent byte units, starting from the second byte unit. If a byte unit does not have enough space to hold the type char member, that member will be allocated to the following byte unit. In this case, if there is only space for 3 bits in the second byte unit, and member d has 4 bits, it will be allocated to the third byte unit.

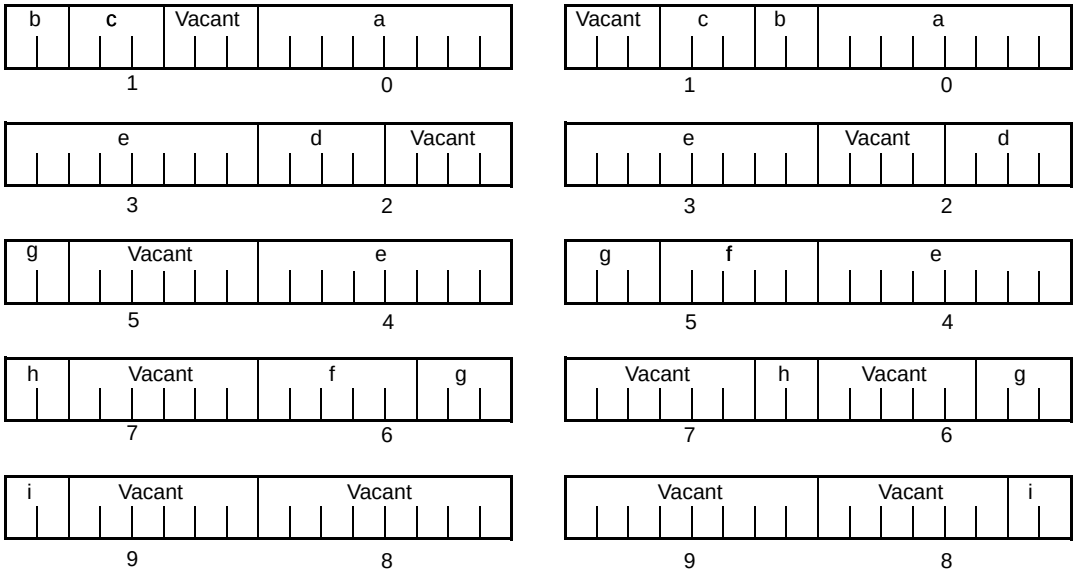


Since member g is a bit field of type unsigned int, it can be allocated across byte boundaries.
Since h is a bit field of type unsigned char, it is not allocated in the same byte unit as the g bit field of type unsigned int, but is allocated in the next byte unit.



Since i is a bit field of type unsigned int, it is allocated in the next word unit.

When the -rc option is specified (to pack the structure members), the above bit field becomes as follows.

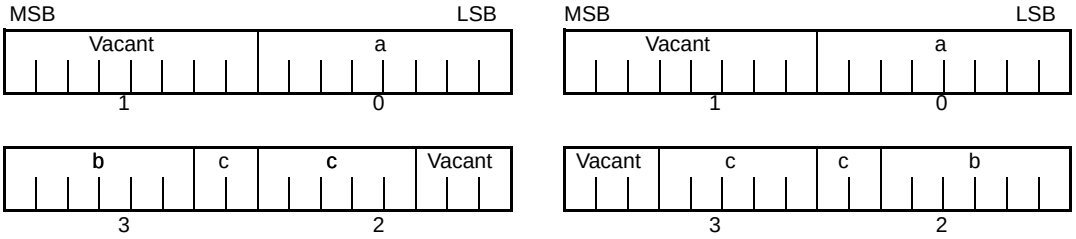


Remark The numbers below the allocation diagrams indicate the byte offset values from the beginning of the structure.

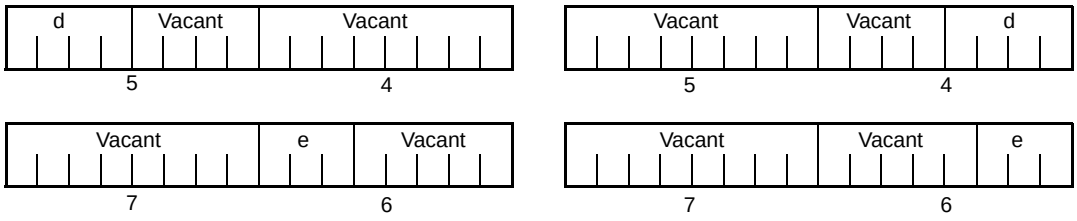
(3) Bit field declaration 3

```
struct  t {
        char            a ;
        unsigned int    b : 6 ;
        unsigned int    c : 7 ;
        unsigned int    d : 4 ;
        unsigned char   e : 3 ;
        unsigned int    f : 10 ;
        unsigned int    g : 2 ;
        unsigned int    h : 5 ;
        unsigned int    i : 6 ;
};
```

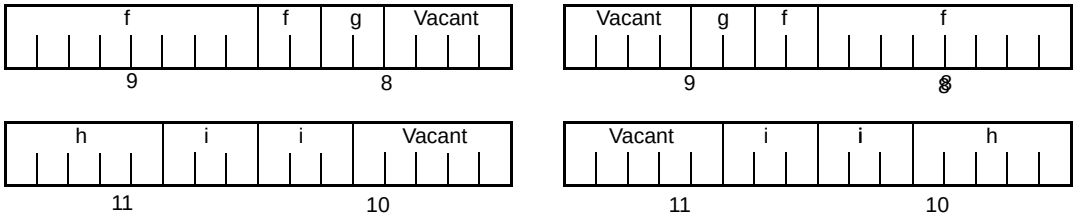
Bit field allocated from the MSB side when the -rb option is specified	Bit field allocated from the LSB side when the -rb option is not specified



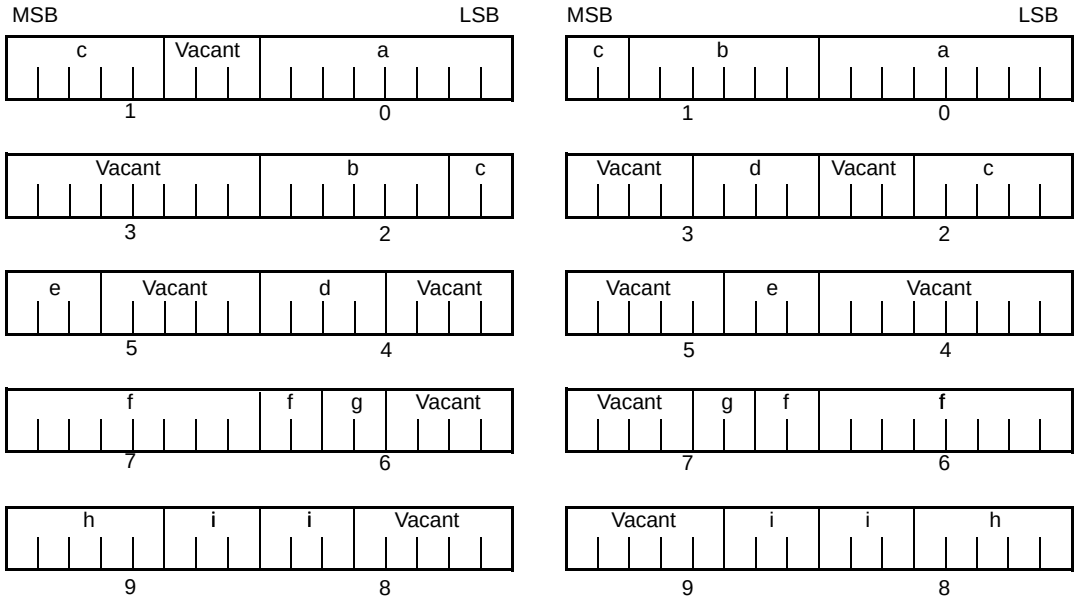
Since b and c are bit fields of type unsigned int, they are allocated from the next word unit.
Since d is also a bit field of type unsigned int, it is allocated from the next word unit.



Since e is a bit field of type unsigned char, it is allocated to the next byte unit.



f and g, and h and i are each allocated to separate word units.
When the -rc option is specified (to pack the structure members), the above bit field becomes as follows.



Remark The numbers below the allocation diagrams indicate the byte offset values from the beginning of the structure.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The source program need not be modified.

(2) From the 78K0R C compiler to another C compiler

- he source program must be modified if the -rb option is used and coding is performed taking the bit field allocation sequence into consideration.

Changing compiler output section name (#pragma section ...)

Changing compiler output section name, and specifying the starting address.

[Function]

- A compiler output section name is changed and a start address is specified.
  If the start address is omitted, the default allocation is assumed. For the compiler output section name and default location, see "3.6 List of Segment Names".
  In addition, the location of sections can be specified by omitting the start address and using the link directive file at the time of link. For the link directives, see "5.1.1 Link directives".
- To change section names @@CALT with an AT start address specified, the callt functions must be described before or after the other functions in the source file.
- If data are described after the #pragma instruction is described, those data are located in the data change section.
  Another change instruction is possible, and if data are described after the rechange instruction, those data are located in the rechange section.
  If data defined before a change are redefined after the change, they are located in the rechanged section.
  Furthermore, this is valid in the same way for static variables (within the function).

[Effect]

- Changing the compiler output section repeatedly in 1 file enables to locate each section independently, so that data can be located in data units to be located independently.

[Usage]

- Specify the name of the section which is to be changed, a new section name, and the start address of the section, by using the #pragma directive as indicated below.
  Describe this #pragma directive at the beginning of the C source.
  The following items can be described before this #pragma directive:
  - Comment
  - Preprocessor directive which does neither define nor refer to a variable or a function

  However, all sections in BSEG and DSEG, and the @@CNST, @@CNSTL section in CSEG can be described anywhere in the C source, and rechange instructions can be performed repeatedly. To return to the original section name, describe the compiler output section name in the changed section.
  Declare as follows at the beginning of the file:

```
#pragma section compiler-output-section-name    new-section-name    [ AT startaddress ]
```

- Of the keywords to be described after #pragma, be sure to describe the compiler output section name in uppercase letters.
  section, AT can be described in either uppercase or lowercase letters, or in combination of those.
- The format in which the new section name is to be described conforms to the assembler specifications (up to 8 letters can be used for a segment name).
- Only the hexadecimal numbers of the C language and the hexadecimal numbers of the assembler can be described as the start address.

(1) Hexadecimal numbers of C language

```
0xn/0xn ... n
0Xn/0Xn ... n
(n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

(2) Hexadecimal numbers of assembler

```
nH/n ... nH
nh/n ... nh
(n = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)
```

The hexadecimal number must start with a numeral.
To example,to express a numeric value with a value of 255 in hexadecimal number, specify zero before F. It is therefore 0FFH.

- For sections other than the @@CNST, @@CNSTL section in CSEG, that is, sections which locate functions, this #pragma instruction cannot be described in other than the beginning of the C source (after the C text is described).If described, a warning is output and the description is ignored.
- If this #pragma instruction is executed after the C text is described, an assembler source file is created without an object module file being created.
- If this #pragma instruction is after the C text is described, a file which contains this #pragma instruction and which does not have the C text (including external reference declarations for variables and functions) cannot be included. This results in an error (see "CODING ERROR EXAMPLE1").
- #include statement cannot be described in a file which executes this #pragma instruction following the C text description. If described, it causes an error (see "CODING ERROR EXAMPLE2").
- If #include statement follows the C text, this #pragma instruction cannot be described after this description. If described, it causes an error (see "CODING ERROR EXAMPLE3").
  But, when a body of C is in the header file, it isn't cause an error.

```
d1.h
        extern int      a;

d2.h
        #define VAR 1

d.c
        #include "d1.h";            // When there is a body of C and it's in #include,
        #include "d2.h"              // #pragma instruction of d.c isn't an error.
        #pragma section @@DATA ??DATA1
```

[Restrictions]

- A section name that indicates a segment for vector table (e.g., @@VECT02, etc.) must not be changed.
- If two or more sections with the same name as the one specifying the AT start address exist in another file, a link error will occur.
- Specify the address within the range from FFE20H to FFEB3H for compiler output section names @@DATS, @@BITS and @@INIS, from 0x80 to 0xbf for @@CALT, from 0x0 to 0xffff for @@CODE and @@BASE, from mirror area for @@CNST, and from 0x0 to 0xffeff for other sections.

[Example]

Section name @@CODEL is changed to CC1 and address 2400H is specified as the start address.
<C source>

```
#pragma section @@CODEL CC1     AT      2400H

void main ( void ) {
        ; Function body
}
```

<Output object of compiler>

```
CC1     CSEG    AT      2400H
_main :
        ; Preprocessing
        ; Function body
        ; Postprocessing
        ret
```

The following is a code example in which the main C code is followed by a #pragma directive.
The contents are allocated in the section following "//".

(1) EXAMPLE1

```
#pragma section @@DATA          ??DATA
int             a1 ;                    // ??DATA
sreg int        b1 ;                    // @@DATS
int             c1 = 1 ;                // @@INIT and @@R_INIT
const int       d1 = 1 ;                // @@CNST
#pragma section @@DATS          ??DATS
int             a2 ;                    // ??DATA
sreg int        b2 ;                    // ??DATS
int             c2 = 1 ;                // @@INIT and @@R_INIT
const int       d2 = 1 ;                // @@CNST
#pragma section @@DATA          ??DATA2
// ??DATA is automatically closed and ??DATA2 becomes valid
int             a3 ;                    // ??DATA2
sreg int        b3 ;                    // ??DATS
int             c3 = 3 ;                // @@INIT and @@R_INIT
const int       d3 = 3 ;                // @@CNST
#pragma section @@DATA          @@DATA
// ??DATA2 is closed and processing returns to the default @@DATA
#pragma section @@INIT          ??INIT
#pragma section @@R_INIT        ??R_INIT

int             a4 ;                    // @@DATA
sreg int        b4 ;                    // ??DATS
// ROMization is invalidated unless both names (@@INIT and @@R_INIT) are changed.  This
// is the user's responsibility.
```

```
int             c4 = 1 ;                // ??INIT and ??R_INIT
const int       d4 = 1 ;                // @@CNST
#pragma section @@INIT          @@INIT
#pragma section @@R_INIT        @@R_INIT
// ??INIT and ??R_INIT are closed and processing returns to the default setting
#pragma section @@BITS          ??BITS
__boolean e4 ;                          // ??BITS
#pragma section @@CNST          ??CNST
char            *const p = "Hello" ;    // p and "Hello" are both ??CNSTT
```

(2) EXAMPLE2

```
#pragma section     @@DATA      ??DATA1
int         a1 ;                        // ??DATA
sreg int    b1 ;                        // @@DATS
int         c1 = 1 ;                    // @@INIT and @@R_INIT
const int   d1 = 1 ;                    // @@CNST
#pragma section     @@DATS      ??DATS
int         a2 ;                        // ??DATA
sreg int    b2 ;                        // ??DATS
int         c2 = 1 ;                    // @@INIT and @@R_INIT
const int   d2 = 1 ;                    // @@CNST
#pragma section     @@DATA      ??DATA2
// ??DATA is automatically closed and ??DATA2 becomes valid
int         a3 ;                        // ??DATA2
sreg int    b3 ;                        // ??DATS
int         c3 = 3 ;                    // @@INIT and @@R_INIT
const int   d3 = 3 ;                    // @@CNST
#pragma section     @@DATA      @DATA
// ??DATA2 is closed and processing returns to the default @@DATA
#pragma section     @@INIT      ??INIT
#pragma section     @@R_INIT    ??R_INIT
// ROMization is invalidated unless both names (@@INIT and @@R_INIT) are changed.  This
//  is the user's responsibility.
int         a4 ;                        // @@DATA
sreg int    b4 ;                        // ??DATS
int         c4 = 1 ;                    // ??INIT and ??R_INIT
const int   d4 = 1 ;                    // @@CNST
#pragma section     @@INIT      @@INIT
#pragma section     @@R_INIT    @@R_INIT
// ??INIT and ??R_INIT are closed and processing returns to the default setting
#pragma section     @@BITS      ??BITS
__boolean   e4 ;                        // ??BITS
#pragma section     @@CNST      ??CNST
char * const p = "Hello" ;              // p and "Hello" are both ??CNSTT
--
#pragma section     @@INIT      ??INIT1
#pragma section     @@R_INIT    ??R_INIT1
```

```
#pragma section     @@DATA      ??DATA1
char        c1 ;
int         i2 ;
#pragma section     @@INIT      ??INIT2
#pragma section     @@R_INIT    ??R_INIT2
#pragma section     @@DATA      ??DATA2
char        c1 ;
int         i2 = 1 ;
#pragma section     @@DATA      ??DATA3
#pragma section     @@INIT      ??INIT3
#pragma section     @@R_INIT    ??R_INIT3
extern char c1 ;                        // ??DATA3
int         i2 ;                        // ??INIT3 and ??R_INIT3
#pragma section     @@DATA      ??DATA4
#pragma section     @@INIT      ??INIT4
#pragma section     @@R_INIT    ??R_INIT4
```

Restrictions when this #pragma directive has been specified after the main C code are explained in the following coding error examples.

(3) CODING ERROR EXAMPLE1

```
a1.h
        #pragma section @@DATA  ??DATA1         // File containing only the #pragma
                                                //  section

a2.h
        extern int      func1( void );s
        #pragma section @@DATA  ??DATA2         // File containing the main C code
                                                // followed by the #pragma directive.

a3.h
        #pragma section @@DATA  ??DATA3         // File containing only the #pragma
                                                // section

a4.h
        #pragma section @@DATA  ??DATA3
        extern int      func2 ( void );         // File that includes the main C code.

a.c
        #include "a1.h"
        #include "a2.h"
        #include "a3.h"         // <- Error
                                // Because the a2.h file contains the main C code
                                // followed by this #pragma directive, file a3.h, which
                                // includes only this #pragma directive, cannot be
                                // included.
        #include "a4.h"
```

(4) CODING ERROR EXAMPLE2

```
b1.h
        const int i ;

b2.h
        const int j ;
        #include "b1.h"         // This does not result in an error since it is not
                                // file (b.c) in which the main C code is followed by
                                // this #pragma directive.

b.c
        const int       k ;
        #pragma section @@DATA  ??DATA1
        #include "b2.h"         // <- Error
                               // Since an #include statement cannot be coded afterward
                               // in file (b.c) in which the main C code is followed by
                                // this #pragma directive.
```

(5) CODING ERROR EXAMPLE3

```
c1.h
        extern int       j ;
        #pragma section @@DATA  ??DATA1 // This does not result in an error since the
                                        //  #pragma directive is included and
                                        // processed before the processing of c3.h.

c2.h
        extern int       k ;
        #pragma section @@DATA  ??DATA2 // <- Error
                                        // This #include statement is specified after
                                        // the main C code in c3.h, and the #pragma
                                        // directive cannot be specified afterward.

c3.h
        #include "c1.h"
        extern int       i ;
        #include "c2.h"
        #pragma section @@DATA  ??DATA3 // <- Error
                                        // This #include statement is specified after
                                        // the main C code, and the #pragma directive
                                        //  cannot be specified afterward.
```

```
c.c
        #include "c3.h"
        #pragma section @@DATA  ??DATA4 // <- Error
                                        // This #include statement is specified after
                                        // the main C code in c3.h, and the #pragma
                                        // directive cannot be specified afterward.
        int     i ;
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The source program need not be modified if the section name change function is not supported.
- To change the section name, modify the source program according to the procedure described in USAGE above.

(1) From the 78K0R C compiler to another C compiler

- Delete or delimit #pragma section ... with #ifdef.
- To change the section name, modify the program according to the specifications of each compiler.

[Cautions]

- A section is equivalent to a segment of the assembler.
- The compiler does not check whether the new section name is in duplicate with another symbol. Therefore, the user must check to see whether the section name is not in duplicate by assembling the output assemble list.
- When the -zf option has been specified, each section name is changed so that the second "@" is replaced with "E".
- If a section name[Note] related to ROMization is changed by using #pragma section, the startup routine must be changed by the user on his/her own responsibility.

Note ROMization-related section name

```
@@R_INIT, @@R_INIS, @@RLINIT, @@INITL, @@INIT, @@INIS
```

Here are examples of changing the startup routine (cstart.asm or cstartn.asm) and termination routine (rom.asm) in connection with changing a section name related to ROMization.

<C source>

```
#pragma section @@R_INIT        RTT1
#pragma section @@INIT          TT1
```

If a section name that stores an external variable with an initial value has been changed by describing #pragma section indicated above, the user must add to the startup routine the initial processing of the external variable to be stored to the new section.

To the startup routine, therefore, add the declaration of the first label of the new section and the portion that copies the initial value, and add the portion that declares the end label to the termination routine, as described below.

RTT1_S and RTT1_E are the names of the first and end labels of section RTT1, and TT1_S and TT1_E are the names of the first and end labels of section TT1.

(1) Changing startup routine cstartx.asm

(a) Add the declaration of the label indicating the end of the section with the changed name

```
  :
#pragma section @@R_INIT    RTT1
#pragma section @@INIT      TT1
EXTRN   RTT1_E, TT1_E          ; Adds EXTRN declaration of RTT1_E and TT1_E
  :
```

(b) Add a section to copy the initial values from the RTT1 section with the changed name to the TT1 section.

```
  :
LDATS1 :
        MOVW    AX, HL
        CMPW    AX, #LOW _?DATS
        BZ      $LDATS2
        MOV     [HL + 0], #0
        INCW    HL
        BR      $LDATS1
LDATS2 :
        MOV     ES, #HIGH RTT1_S
        MOV     HL, #LOWW RTT1_S
        MOV     DE, #LOWW TT1_S
LTT1 :
        MOVW    AX, HL
        CMPW    AX, #LOWW TT1_E
        BZ      $LTT2
        MOV     A, ES : [HL]
        MOV     [DE], A
        INCW    HL
        INCW    DE
        BR      $LTT1
LTT2 :
;
        CALL    !!_main                 ; main ( );
        CLRW    AX
        CALL    !!_exit                 ; exit ( 0 );
        BR      $$
;
```

Adds section to copy the initial values from the RTT1 section to the TT1 section

(c) Set the label of the start of the section with the changed name.

```
  :
@@R_INIT        CSEG    UNIT64KP
_@R_INIT :
@@R_INIS        CSEG    UNIT64KP
_@R_INIS :
@@INIT          DSEG
_@INIT :
@@DATA          DSEG
_@DATA :
@@INIS          DSEG    SADDRP
_@INIS :
@@DATS          DSEG    SADDRP
_@DATS :

RTT1            CSEG    UNIT64KP        ; Indicates the start of the RTT1 section
RTT1_S :                                ; Adds the label setting
TT1             DSEG    BASEP           ; Indicates the start of the TT1 section
TT1_S :                                 ; Adds the label setting

@@CODEL         CSEG
@@CALT          CSEG    CALLT0
@@CNST          CSEG    MIRRORP
@@BITS          BSEG
;
                END
```

(2) Changing termination routine rom.asm

Caution Don't change the object module name "@rom" and "@rome".

(a) Add the declaration of the label indicating the end of the section with the changed name

```
NAME            @rom
;
PUBLIC          _?R_INIT, _?R_INIS
PUBLIC          _?INIT, _?DATA, _?INIS, _?DATS

PUBLIC          RTT1_E, TT1_E          ; Adds RTT1_E and TT1_E

;
@@R_INIT        CSEG    UNIT64KP
_?R_INIT :
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT          DSEG
_?INIT :
@@DATA          DSEG
_?DATA :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS
        :
```

(b) Setting the label indicating the end

```
  :
RTT1    CSEG    UNIT64KP        ; Adds the label setting indicating the end of the
                                ; RTT1 section.
RTT1_E :                        ; Adds the label setting

TT1     DSEG    BASEP           ; Adds the label setting indicating the end of the
                                ; TT1 section.
TT1_E :                         ; Adds the label setting

;
        END
```

Binary constant (0bxxx)

The compiler supports the 0bxxx notation for expressing binary constants in C source code.

[Function]

- Describes binary constants to the location where integer constants can be described.

[Effect]

- Constants can be described in bit strings without being replaced with octal or hexadecimal number. Readability is also improved.

[Usage]

- Describe binary constants in the C source.
  The following shows the description method of binary constants.

```
0b      binary-number
0B      binary-number
```

Remark Binary number: either "0" or "1".

- A binary constant has 0b or 0B at the start and is followed by the list of numbers 0 or 1.
- The value of a binary constant is calculated with 2 as the base.
- The type of a binary constant is the first one that can express the value in the following list.

Subscripted binary number:	int, unsigned int, long int, unsigned long int
Subscripted u or U:	unsigned int, unsigned long int
Subscripted l or L:	long int, unsigned long int
Subscripted u or U and subscripted l or L with:	unsigned long int

[Example]

<C source>

```
unsigned        i ;

i = 0b11100101 ;
```

Output object of compiler is the same as the following case.

```
unsigned        i ;

i = 0xE5 ;
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed.

(2) From the 78K0R C compiler to another C compiler

- Modifications are needed to meet the specification of the compiler if the compiler supports binary constants.
- Modifications into other integer formats such as octal, decimal, and hexadecimal are needed if the compiler does not support binary constants.

Module name changing function (#pragma name)

The module name of an object can be changed to any name in C source code.

[Function]

- Outputs the first 254 letters of the specified module name to the symbol information table in a object module file.
- Outputs the first 254 letters of the specified module name to the assemble list file as symbol information (MOD_NAM) when the -g2 option is specified and as NAME pseudo instruction when the -ng option is specified.
- If a module name with 255 or more letters are specified, a warning message is output.
- If unauthorized letters are described, an error will occur and the processing is aborted.
- If more than one of this #pragma directive exists, a warning message is output, and whichever described later is enabled.

[Effect]

- The module name of an object can be changed to any name.

[Usage]

- The following shows the description method.

```
#pragma name    module-name
```

A module name must consist of the characters that the OS authorizes as a file name except "(", ")", and kanji (2-byte character).
Upper/lowercase is distinguished.

[Example]

```
#pragma name    module1
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed if the compiler does not support the module name changing function.
- To change a module name, modification is made according to USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #pragma name ... is deleted or sorted by #ifdef.
- To change a module name, modification is needed depending on the specification of each compiler.

Rotate function (#pragma rot)

Outputs the code that rotates the value of an expression to the object with direct inline expansion.

[Function]

- Outputs the code that rotates the value of an expression to the object with direct inline expansion instead of function call and generates an object file.
- If there is not a #pragma directive, the rotate function is regarded as an ordinary function.

[Effect]

- Rotate function is realized by the C source or ASM description without describing the processing to perform rotate.

[Usage]

- Describe in the source in the same format as the function call.
  There are the following 4 function names.

```
rorb, rolb, rorw, rolw
```

(1) unsigned char rorb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Rotates *x* to right for *y* times.

(2) unsigned char rolb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Rotates *x* to left for *y* times.

(3) unsigned int rorw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
Rotates *x* to right for *y* times.

(4) unsigned int rolw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
Rotates *x* to left for *y* times.

- Declare the use of the function for rotate by the #pragma rot directive of the module.
  However, the followings can be described before #pragma rot.
  - Comments
  - Other #pragma directives
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Keywords following #pragma can be described in either uppercase or lowercase letters.

[Example]

<C source>

```
#pragma rot
unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned char   c ;

void main ( void ) {
        c = rorb ( a, b );
}
```

<Output assembler source>

```
        mov     x, !_b
        mov     a, !_a
L0003 :
        ror     a, 1
        dec     x
        bnz     $L0003
```

[Restrictions]

- The function names for rotate cannot be used as the function names.
- The function names for rotate must be described in lowercase letters. If the functions for rotate are described in uppercase letters, they are handled as ordinary functions.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modification is not needed if the compiler does not use the functions for rotate.
- To change to functions for rotate, modifications are made according to USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #pragma rot statement is deleted or sorted by #ifdef.
- To use as a function for rotate, modification is needed depending on the specification of each compiler (#asm, #endasm or asm( ) ; , etc.).

Multiplication function (#pragma mul)

Outputs the code that multiplies the value of an expression to the object with direct inline expansion.

[Function]

- Outputs the code that multiplies the value of an expression to the object with direct inline expansion instead of function call and generates an object file.
- If there is not a #pragma directive, the multiplication function is regarded as an ordinary function.

[Effect]

- The codes utilizing the data size of input/output of the multiplication instruction are generated. Therefore, the codes with faster execution speed and smaller size than the description of ordinary multiplication expressions can be generated.

[Usage]

- Describe in the same format as that of function call in the source.
  The following shows list of multiplication function.

```
mulu
```

(1) unsigned int mulu (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Performs unsigned multiplication of *x* and *y*.

- Declare the use of functions for multiplication by #pragma mul directive of the module.
  However, the followings can be described before #pragma mul.
  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate definition/reference of variables and definition/reference of functions
- Keywords following #pragma can be described in either uppercase or lowercase letters.

[Restrictions]

- The function for multiplication cannot be used as the function names (when #pragma mul is declared).
- The function for multiplication must be described in lowercase letters. If they are described in uppercase letters, they are handled as ordinary functions.

[Example]

<C source>

```
#pragma mul

unsigned char   a = 0x11 ;
unsigned char   b = 2 ;
unsigned int    i ;

void main ( void ) {
        i = mulu ( a, b );
}
```

<Output object of compiler>

```
mov     x, !_b
mov     a, !_a
mulu    x
movw    !_i, ax
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed if the compiler does not use the functions for multiplication.
- To change to functions for multiplication, modification is made according to USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #pragma mul statement is deleted or sorted by #ifdef. Function names for multiplication can be used as the function names.
- To use as functions for multiplication, modification is needed depending on the specification of each compiler (#asm, #endasm or asm( ) ; , etc.).

Division function (#pragma div)

Outputs the code that divides the value of an expression to the object.

[Function]

- Outputs the code that divides the value of an expression to the object.
- If there is not a #pragma directive, the function for division is regarded as an ordinary function.

[Effect]

- Codes that are compatible with the 78K0 C compiler and utilize the data size of the division instruction I/O are generated. Therefore, codes with faster execution speed and smaller size than the description of ordinary division expressions can be generated.

[Usage]

- Describe in the same format as that of function call in the source.
  There are the following 2 functions for division.

```
divuw, moduw
```

(1) unsigned int divuw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
Performs unsigned division of *x* and *y* and returns the quotient.

(2) unsigned char moduw (*x* , *y*) ;
unsigned int *x* ;
unsigned char *y* ;
Performs unsigned division of *x* and *y* and returns the remainder.

- Declare the use of the function for divisions by the #pragma div directive of the module.
  However, the followings can be described before #pragma div.
  - Comments
  - Other #pragma directives
  - Preprocessing directives which do not generate definition/reference of variables and definition/ reference of functions
- Keywords following #pragma can be described in either uppercase or lowercase letters.

[Restrictions]

- The division functions are not expanded inline, but are called by the library.
- The function names for division cannot be used as the function names.
- The function names for division must be described in lowercase letters. If they are described in uppercase letters, they are handled as ordinary functions.

[Example]

<C source>

```
#pragma div

unsigned int    a = 0x1234 ;
unsigned char   b = 0x12 ;
unsigned char   c ;
unsigned int    i ;

void main ( void ) {
        i = divuw ( a, b );
        c = moduw ( a, b );
}
```

<Output object of compiler>

```
mov     c, !_b
movw    ax, !_a
call    !@@divuw
movw    !_i, ax
mov     c, !_b
movw    ax, !_a
call    !@@divuw
mov     a, c
mov     !_c, a
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modification is not needed if the compiler does not use the functions for division.
- To change to functions for division, modifications are made according to USAGE above.

(2) From the 78K0R C compiler to another C compiler

- #pragma div statement is deleted or sorted by #ifdef. The function names for division can be used as the function name.
- To use as a function for division, modification is needed depending on the specification of each compiler (#asm, #endasm or asm( ) ; , etc.).

BCD operation function (#pragma bcd)

Outputs the code that performs a BCD operation on the expression value in an object by direct inline expansion.

[Function]

- Outputs the code that performs a BCD operation on the expression value in an object by direct inline expansion rather than by function call, and generates an object file.
  However, bcdtob, btobcde, bcdtow, wtobcd and btobcd function are not developed inline.
- If there are no #pragma directives, the function for BCD operation is regarded as an ordinary function.

[Effect]

- Even if the process of the BCD operation is not described, the BCD operation function can be realized by the C source or ASM statements.

[Usage]

- The same format as that of a function call is coded in the source.
  There are 13 types of function name for BCD operation, as listed below.

(1) unsigned char adbcdb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Decimal addition is carried out by the BCD adjustment instruction.

(2) unsigned char sbbcdb (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Decimal subtraction is carried out by the BCD adjustment instruction.

(3) unsigned int adbcdbe (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Decimal addition is carried out by the BCD adjustment instruction (with result expansion).

(4) unsigned int sbbcdbe (*x* , *y*) ;
unsigned char *x* ;
unsigned char *y* ;
Decimal subtraction is carried out by the BCD adjustment instruction (with result expansion).
If a borrow occurs, the high-order digits are set to 0x99.

(5) unsigned int adbcdw (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
Decimal addition is carried out by the BCD adjustment instruction.

(6) unsigned int sbbcdw (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
Decimal subtraction is carried out by the BCD adjustment instruction.

(7) unsigned long adbcdwe (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
Decimal addition is carried out by the BCD adjustment instruction (with result expansion).

(8) unsigned long sbbcdwe (*x* , *y*) ;
unsigned int *x* ;
unsigned int *y* ;
Decimal subtraction is carried out by the BCD adjustment instruction (with result expansion).
If a borrow is occurred, the higher digits are set to 0x9999.

(9) unsigned char bcdtob (*x*) ;
unsigned char *x* ;
Values in decimal number are converted to binary number values.

(10) unsigned int btobcde (*x*) ;
unsigned char *x* ;
Values in binary number are converted to decimal number values.

(11) unsigned int bcdtow (*x*) ;
unsigned int *x* ;
Values in decimal number are converted to binary number values.

(12) unsigned int wtobcd (*x*) ;
unsigned int *x* ;
Values in decimal number are converted to binary number values.
However, if the value of x exceeds 10000, 0xffff is returned.

(13) unsigned char btobcd (*x*) ;
unsigned char *x* ;
Values in decimal number are converted to those in binary number.
However, the overflow is discarded.

- Use of functions for BCD operation is declared by the module's #pragma bcd directive. The following items, however, can be coded before #pragma bcd.
  - Comments
  - Other #pragma directives
  - Preprocessing directives that do not generate definitions/sreferences of variables or function definitions/ references
- Either uppercase or lowercase letters can be used for keywords described after #pragma.

[Restrictions]

- BCD operation function names cannot be used as function names.
- The BCD operation function is coded in lowercase letters. If uppercase letters are used, these functions are regarded as an ordinary functions.

[Example]

<C source>

```
#pragma  bcd

unsigned char   a = 0x12 ;
unsigned char   b = 0x34 ;
unsigned char   c ;

void main ( void ) {
        c = adbcdb ( a, b );
        c = sbbcdb ( b, a );
}
```

<Output object of compiler>

```
mov     a, !_a
add     a, !_b
add     a, !BCDADJ
mov     !_c, a
mov     a, !_b
sub     a, !_a
sub     a, !BCDADJ
mov     !_c, a
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Corrections are not needed if functions for the BCD operations are not used.
- To change another function to the function for BCD operation, use the description above.

(1) From the 78K0R C compiler to another C compiler

- The #pragma bcd statements are either deleted or separated by #ifdef. A BCD operation function name can be used as a function name.
- If using "pragma bcd" as a BCD operation function, the changes to the program source must conform to the C compiler's specifications (#asm, #endasm or asm( ); etc.).

Data insertion function (#pragma opc)

Inserts constant data into the current address.

[Function]

- Inserts constant data into the current address.
- When there is not a #pragma directive, the function for data insertion is regarded as an ordinary function.

[Effect]

- Specific data and instruction can be embedded in the code area without using the ASM statement.
  When ASM is used, an object cannot be obtained without the intermediary of assembler. On the other hand, if the data insertion function is used, an object can be obtained without the intermediary of assembler.

[Usage]

- Describe using uppercase letters in the source in the same format as that of function call.
- The function name for data insertion is __OPC.

(1) void __OPC (unsigned char *x* , ...) ;

Insert the value of the constant described in the argument to the current address.
Arguments can describe only constants.

- Declare the use of functions for data insertion by the #pragma opc directive.
  However, the followings can be described before #pragma opc.
  - Comments
  - Other #pragma directives
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Keywords following #pragma can be described in either uppercase or lowercase letters.

[Restrictions]

- The function names for data insertion cannot be used as the function names (when #opc is specified).
- __OPC must be described in uppercase letters. If they are described in lowercase letters, they are handled as ordinary functions.

[Example]

<C source>

```
#pragma opc

void main ( void ) {
        __OPC ( 0xA7 );
        __OPC ( 0x51, 0x12 );
        __OPC ( 0x30, 0x34, 0x12 );
}
```

<Output object of compiler>

```
_main :
; line  4 : __OPC ( 0xA7 );
        DB      0AFH
; line  5 : __OPC ( 0x51, 0x12 );
        DB      051H
        DB      012H
; line  6 : __OPC ( 0x30, 0x34, 0x12 );
        DB      030H
        DB      034H
        DB      012H
; line  7 : }
        ret
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modification is not needed if the compiler does not use the functions for data insertion.
- To change to functions for data insertion, use the USAGE above.

(2) From the 78K0R C compiler to another C compiler

- The #pragma opc statement is deleted or delimited by #ifdef. Function names for data insertion can be used as function names.
- To use as a function for data insertion, changes to the program source must conform to the specification of the C compiler (#asm, #endasm or asm( ) ; , etc.).

Interrupt handler for RTOS (#pragma rtos_interrupt ...)

The interrupt handler for RX78K0R can be described.

[Function]

- Interprets the function name specified with the #pragma rtos_interrupt directive as the interrupt handler for the 78K0R RTOS RX78K0R.
- Registers the address of the described function name to the interrupt vector table for the specified interrupt request name.
- The interrupt handler for RTOS generates codes in the following order.

(1) Calls kernel symbol __kernel_int_entry using call !!addr20 instruction

(2) Saves the saddr area used by compiler

(3) Secures the local variable area (only when there is a local variable)

(4) The function body

(5) Releases the local variable area (only when there is a local variable)

(6) Restores the saddr area used by compiler

(7) Unconditionally jumps to label _ret_int using br !!addr20 instruction

[Effect]

- The interrupt handler for RTOS can be described in the C source level.
- Because the interrupt request name is identified, the address of the vector table does not need to be identified.

[Usage]

- The interrupt request name, function name is specified by the #pragma directive.

```
#pragmaΔrtos_interrupt[Δinterrupt-request-nameΔfunction-name]
```

- This #pragma directive is described at the start of the C source.
- The following can be described before the #pragma directive.
  - Comments
  - Preprocessing directives which do not generate definition/reference of variables and definition/reference of functions
- Of the keywords to be described following #pragma, the interrupt request name must be described in uppercase letters. The other keywords can be described either in uppercase or lowercase letters.

[Restrictions]

- When the -zf option is not specified, interrupt handler for RTOS are allocated to the area between C0H and 0FFFFH, regardless of the memory model.
  When the -zf option is specified, interrupt functions are allocated in accordance with the memory model. In addition, specification of allocation area by specifying __near or __far is also enabled.
- Interrupt request names are described in uppercase letters.
- Software interrupts and non-maskable interrupts cannot be specified for the interrupt request names, if specified so, an error will occur.
- Interrupt requests are double-checked in one module units only.
- The interrupt handler for RTOS cannot specify callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf.
  __far can be specified only when the -zf option is specified.
- ret_int/_kernel_int_entry cannot be used for the function names.

[Example]

<C source>

```
#pragma rtos_interrupt  INTP0   intp

int     i ;

void intp ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        func ();
}
```

<Output object of compiler>

```
@@BASE          CSEG    BASE
_intp :
        call    !!__kernel_int_entry
        movw    ax, _@RTARG0    ; Saves saddr area used by the compiler
        push    ax              ;
        movw    ax, _@RTARG2    ;
        push    ax              ;
        movw    ax, _@RTARG4    ;
        push    ax              ;
        movw    ax, _@RTARG6    ;
        push    ax              ;
        movw    ax, _@SEGAX     ;
        push    ax              ;
        movw    ax, _@SEGDE     ;
        push    ax              ;
        subw    sp, #06H        ; Secures the local variable area
        movw    hl, sp
; line 5 :      int     a[3] ;
; line 6 :      a[0] = 1;
        onew    ax
        movw    [hl], ax       ; a
```

```
; line 7 :      func ();
         call    !!_func
; line 8 :      }
        addw    sp, #06H        ; Releases the local variable area
        pop     ax              ; Restores saddr area used by the compiler
        movw    _@SEGDE, ax     ;
        pop     ax              ;
        movw    _@SEGAX, ax     ;
        pop     ax              ;
        movw    _@RTARG6, ax    ;
        pop     ax              ;
        movw    _@RTARG4, ax    ;
        pop     ax              ;
        movw    _@RTARG2, ax    ;
        pop     ax              ;
        movw    _@RTARG0, ax    ;
        br      !!_ret_int

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed if the compiler does not support the interrupt handler for RTOS.
- To change to interrupt handler for RTOS, use the USAGE above.

(2) From the 78K0R C compiler to another C compiler>

- Handled as an ordinary function if #pragma rtos_interrupt specification is deleted.
- To use as an interrupt handler for ROTS, changes to the source program must conform to the specification of the C compiler.

Interrupt handler qualifier for RTOS (__rtos_interrupt)

The setting of the vector and the description of the interrupt handler for RX78K0R can be described in separate files.

[Function]

- The function declared with the __rtos_interrupt qualifier is interpreted as an interrupt handler for RTOS. For details on registers used with interrupt handler for RTOS and saving and restoring of saddr, refer to "Interrupt handler for RTOS (#pragma rtos_interrupt ...)".

[Effect]

- The setting of the vector table and the definition of the interrupt handler function for RTOS can be described in separate files.

[Usage]

- __rtos_interrupt is added to the qualifier of the interrupt handler for RTOS.

```
__rtos_interrupt        void func ( )        {processing}
```

[Restrictions]

- When the -zf option is not specified, interrupt handler for RTOS are allocated to the area between C0H and 0FFFFH, regardless of the memory model. When the -zf option is specified, interrupt functions are allocated in accordance with the memory model. In addition, specification of allocation area by specifying __near or __far is also enabled.
- The interrupt handler for RTOS cannot specify callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf. __far can be specified only when the -zf option is specified.
- ret_int/__kernel_int_entry cannot be used for the function names.

[Cautions]

- Vector addresses cannot be set only with declaration of this qualifier.
  The setting of the vector address must be performed separately with the #pragma directive, assembler description, etc.
- When the interrupt handler for RTOS is defined in the same file as the one in which the #pragma rtos_interrupt ... is specified, the function name specified with #pragma rtos_interrupt is judged as an interrupt handler for RTOS even if this qualifier is not described.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed if the compiler does not support interrupt handler for RTOS.
- To change to interrupt handler for RTOS, use the USAGE above.

(2) From the 78K0R C compiler to another C compiler

- Changes can be made by #define (For the details, refer to "3.3.5 C source modifications").
  By these changes, interrupt handler qualifiers for RTOS are handled as ordinary variables.
- To use as an interrupt handler for RTOS, modification is needed depending on the specification of each compiler.

Task function for RTOS (#pragma rtos_task)

The function names specified with #pragma rtos_task are interpreted as the tasks for RX78K0R.

[Function]

- The function names specified with #pragma rtos_task are interpreted as the tasks for RTOS.
- In the case the function name is specified, if the entity definition is not in the same file, an error will occur.
- The preprocessing of the task function for RTOS does not save the registers for frame pointer/register variables. The postprocessing is not output.
- RTOS system call ext_tsk is always called at the end of #pragma rtos_task.
- The following RTOS system call calling function can be used.

  void ext_tsk (void) ;

  Calls RTOS system call ext_tsk.

  When ext_tsk is, however, called in the ext_tsk entity definition, interrupt function, interrupt handler for RTOS, an error will occur.
- RTOS system call ext_tsk is called using the br !!addr20 instruction. If ext_tsk is issued at the end of an ordinary function, the epilogue is not output.
- A task function can be coded without arguments specified, or with only one argument of up to 4 bytes specified, but no return values can be specified.

  An error will be occur if two or more arguments are specified, an argument of 5 bytes or longer is specified, or a return value is specified.

[Effect]

- The task function for RTOS can be described in the C source level.
- The saving and postprocessing of the register frame pointer/register variable are not output, so the code efficiency is improved.

[Usage]

- Specifies the function name for the following #pragma directives.

```
#pragmaΔrtos_task[Δtask-function-name]
```

- The #pragma directives are described at the start of the C source.

  However, the followings can be described before the #pragma directive.
  - Comments
  - Preprocessing directives which do not generate definition/reference of variables and definition/ reference of functions
- Keywords following #pragma can be described either in uppercase or lowercase letters.

[Restrictions]

- The task function for RTOS cannot specify the callt/__callt/__interrupt/__interrupt_brk/__flash/__flashf. __far can be specified only when the -zf option is specified.
- The task function for RTOS cannot be called in the same manner as the ordinary functions.
- RTOS system call calling function name ext_tsk cannot be used for function names.
- If #pragma rtos_task is not written to the C source, ext_tsk is not interpreted as a system call for RTOS. Consequently, the following error will not be output even if ext_tsk is called from an RTOS interrupt handler.

  E0778: Cannot call ext_tsk in interrupt function

  Workarounds:
  - Clearly specify the use of the task function, by specifying #pragma rtos_task.
  - Do not call ext_tsk from RTOS interrupt handlers.

[Example]

<C source>

```
#pragma rtos_task       func
#pragma rtos_task       func2

void func ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        ext_tsk ( );
}

void func2 ( int x ) {
        int     a[3] ;
        a[0] = 1 ;
}

void func3 ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        ext_tsk ( );
}

void func4 ( void ) {
        int     a[3] ;
        a[0] = 1 ;
        if ( a[0] ) {
                ext_tsk ( );
        }
}
```

<Output object of compiler>

```
@@CODEL CSEG
_func :
        subw    sp, #06H        ; Frame pointers are saved
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        br      !!_ext_tsk      ; Calling of ext_tsk by writing ext_tsk function
        br      !!_ext_tsk      ; Calling of ext_tsk always output by task function
                                ; Epilogue is not output
_func2 :
        push    ax              ; Frame pointers are not saved
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        br      !!_ext_tsk      ; Calling of ext_tsk always output by task function
                                ; Epilogue is not output
_func3 :
        push    hl              ; Frame pointers are saved
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        br      !!_ext_tsk      ; Epilogue is output if ext_tsk is called in the middle of
                                ; a function
_func4 :
        push    hl              ; Frame pointers are saved
        subw    sp, #06H
        movw    hl, sp
        onew    ax
        movw    [hl], ax    ; a
        clrw    bc
        cmpw    ax, bc
        skz
        br      !!_ext_tsk      ; Epilogue is output if ext_tsk is called
        addw    sp, #06H        ; in the middle of a function
        pop     hl
        ret
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modifications are not needed if the compiler does not support the task function for RTOS.
- To change to the task function for RTOS, use the USAGE above.

(2) From the 78K0R C compiler to another C compiler

- If #pragma rtos_task specification is deleted, RTOS task function is used as an ordinary function.
- To use as RTOS task function, changes to the program source must conform to the specification of the C compiler.

Flash area allocation method (-zf)

Enables locating a program in the flash area compiling with specifying the -zf option. Enables using function linking with a boot area object created without specifying the -zf option.

Caution This function enables the flash memory rewriting function of devices.

[Function]

- Generates an object file located in the flash area.
- External variables in the flash area cannot be referred to from the boot area.
- External variables in the boot area can be referred to from the flash area.
- The same external variables and the same global functions cannot be defined in a boot area program and a flash area program.

[Effect]

- Enables locating a program in the flash area.
- Enables using function linking with a boot area object created without specifying the -zf option.

[Usage]

- Specify the -zf option during compiling.

[Restrictions]

- Use startup routines or library for the flash area.

Flash area branch table and flash area allocation (#pragma ext_table)

The #pragma directive specify the starting address of the flash branch table. A startup routine and interrupt function can be located in the flash area and a function calls can be performed from the boot area to the flash area.

Caution This function enables the flash memory rewriting function of devices.

[Function]

- The -zf options determines the first address of the branch table for the startup routine, the interrupt function, or the function call from the boot area to the flash area.
- 64 addresses from the first address of the branch table are dedicated for interrupt functions (including startup routine), and each of them occupies 4 bytes of area.
- The branch tables for ordinary functions are normally allocated after the "first address of the branch table + 4 * 64". Each of the branch tables occupies 4 bytes of area. See "Function of function call from boot area to flash area (#pragma ext_func)" for more information about ext_func ID values.
- The -zz options determine the starting address of the branch table.
- When only the -zt option is specified, the -zz option is regarded as having the same value.
- When only the -zz option is specified, the -zt option is regarded as having the same value.

[Effect]

- A startup routine and interrupt function can be located in the flash area.
- A function calls can be performed from the boot area to the flash area.

[Usage]

- Use the -zt option as follows to specify the starting address of the flash branch table.

```
-ztxxxxxh : xxxxx = 0c0h to 0edfffh[Note]
```

- Use the -zz option as follows to specify the starting address of the flash branch table.

```
-zzxxxxxh : xxxxx = 0c0h to 0edfffh[Note]
```

Note The address varies on different devices.

[Restrictions]

- The range of addresses that may be specified as the starting address of the flash branch table is 0C0H to 0EDFFFH. (However, the 0EDFFFH varies according to the target device.)
- Either the -zf option must be specified when the source program contains a #pragma ext_func, and when the the -zf option is specified and the program contains a #pragma vect, #pragma interrupt or a #pragma rtos_interrupt directive. An error occurs if the -zz or -zt option is not specified.
- 2000H is the default starting address of the interrupt service routine library vector table (_@vect00 to _@vect7e). The default of the start address of the branch table in the interrupt vector library is 2000H.
- The linker option -zb also specifies the starting address of the flash branch table. Always specify the same address for the linker option -zb and the starting address of the flash area. An error occurs if the addresses do not agree.
- An error occurs if the allocation address of the flash branch table is smaller than the starting address of the flash branch table
- The -zt or -zz option must be used to specify the allocation address of the flash area and the flash branch table if you are creating a program to be located in the boot area or the flash area.
- An error occurs when modules compiled with different -zt or -zz address specifications are linked.

- Pointers to ROM data are forcibly handled as far pointers when the ROM data of the boot area or flash area cannot be located in a near area. (See the NOTES below.) Consequently, in the small and medium models, the suffix "_f" must be added after the library function name when calling a standard library function that takes a (const *) argument (warning W0072 is always output).
  The following standard library functions take (const *) arguments.
  sprintf/sscanf/printf/scanf/vprintf/vsprintf/puts/atoi/atol/strtol/stltoul/atof/strtod/bsearch/qsort/memcpy/memmove/strcpy/strncpy/strcat/strncat/memcmp/strcmp/strncmp/memchr/strchr/strcspn/strpbrk/strrchr/strspn/strstr/strtok/strlen/strcoll/strxfrm

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- To specify the first address of the flash area branch table, change the address in accordance with USAGE above.

(2) From the 78K0R C compiler to another C compiler

- To specify the first address of the flash area branch table, the following change is required.

[Cautions]

- The starting address of the flash branch table and the starting address of the mirror area affect the handling of near/far specifications. If near/far area specifications are different from the actual memory layout, warnings W0070 and W0071 are issued once only, at the time of command line analysis.
- When the starting address of the flash branch table is within the mirror area, and within 64 KB: near/far area specifications are followed without change. (See "Figure 3-3. Memory Map Example 1".)
- When the starting address of the flash branch table is within the mirror area, and not within 64 KB: flash area functions are located in a far area. (See "Figure 3-4. Memory Map Example 2".)
- When the starting address of the flash branch table is above the end address of the mirror area, and within 64 KB: flash area ROM data is located in a far area. (See "Figure 3-5. Memory Map Example 3".)
- When the starting address of the flash branch table is above the end address of the mirror area, and not within 64 KB: flash area functions are located in a far area, and flash ROM data is located in a far area. (See "Figure 3-6. Memory Map Examplee 4".)
- When the starting address of the flash branch table is below the starting address of the mirror area, and within 64 KB: boot area ROM data is located in a far area. (See "Figure 3-7. Memory Map Example 5".)
- When the starting address of the flash branch table is below the starting address of the mirror area, and not within 64 KB: boot area ROM data is located in a far area, and flash area ROM data is located in a far area. (See "Figure 3-8. Memory Map Example 6".)
- When boot area or flash area ROM data cannot be placed in a near area: pointers to ROM data are always far. Consequently, small and medium model programs no longer conform to the ANSI standard. When the strict ANSI conformance option -za is specified, warning W0073 is issued.
- When boot area or flash area ROM data cannot be placed in a near area, or when flash area functions cannot be placed in a near area: the following restrictions apply.

Table 3-13. Handling of ROM Data When There Is No Mirror of Boot Area

Area	Definition	Extern Declaration	Object Pointed to by Pointer
Boot	Always far	Always far	Always far
Flash	near or far	Always far	Always far

Table 3-14. Handling of ROM Data When There Is No Mirror of Flash Area

Area	Definition	Extern Declaration	Object Pointed to by Pointer
Boot	near or far	near or far	Always far
Flash	Always far	Always far	Always far

Table 3-15. Handing of Functions When Start of Flash Area Is Not Within 64 KB

Area	Definition	Extern Declaration	Object Pointed to by Pointer
Boot	near or far	near or far **Note**	Always far
Flash	Always far	Always far	Always far

Note Functions specified by #pragma ext_func reside in flash memory, so they are always far.

Figure 3-3. Memory Map Example 1

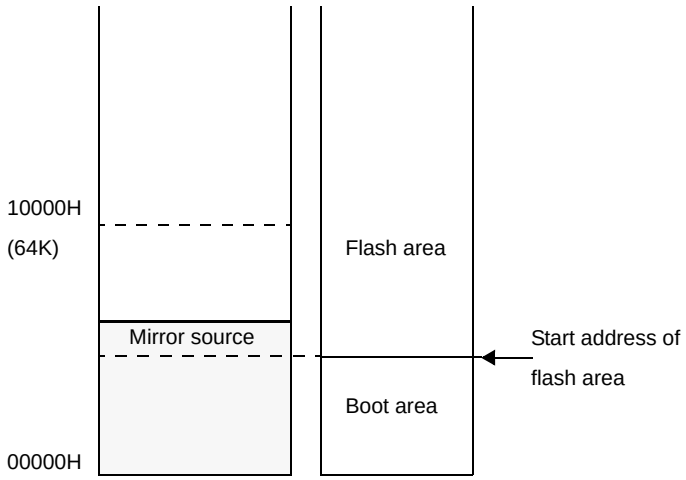


Figure 3-4. Memory Map Example 2

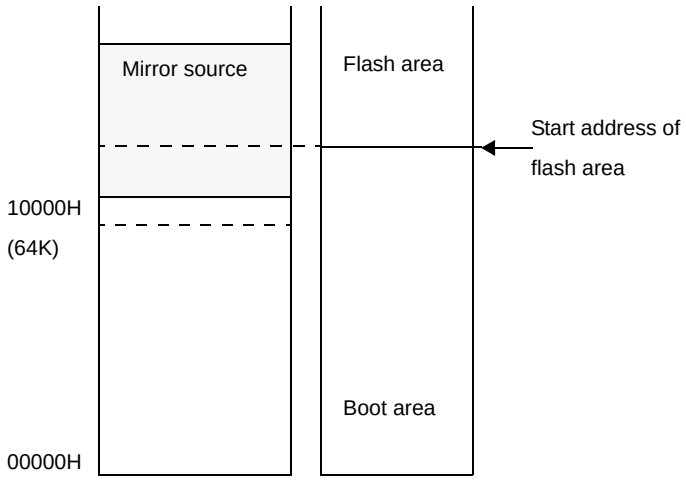


Figure 3-5. Memory Map Example 3

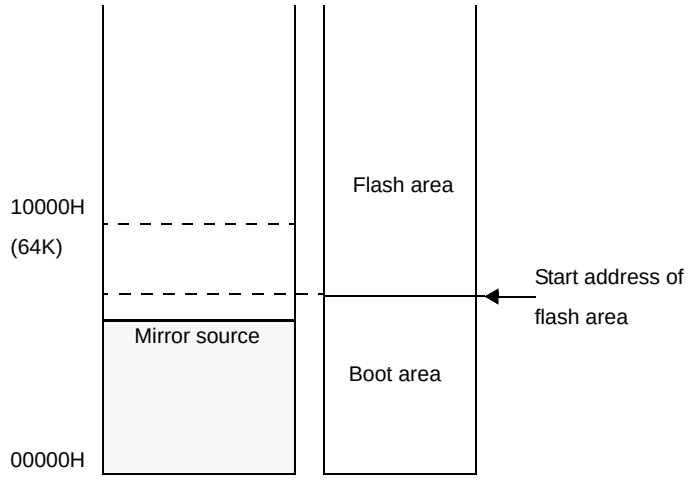


Figure 3-6. Memory Map Examplee 4

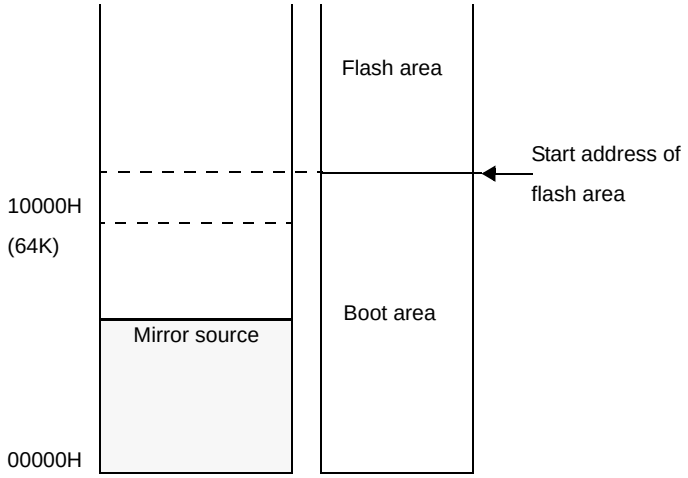


Figure 3-7. Memory Map Example 5

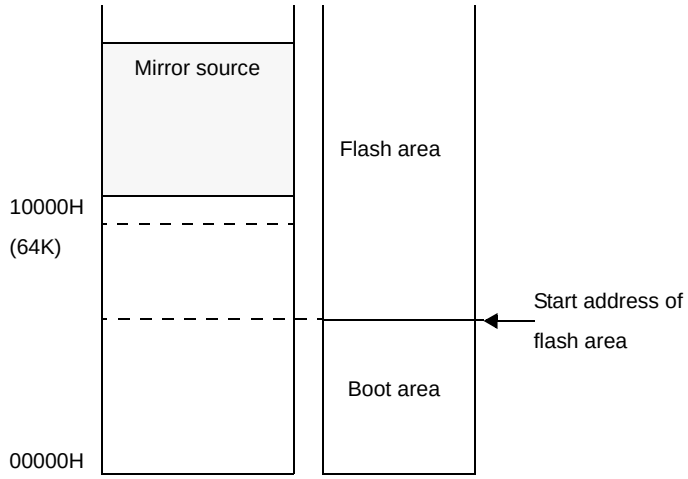
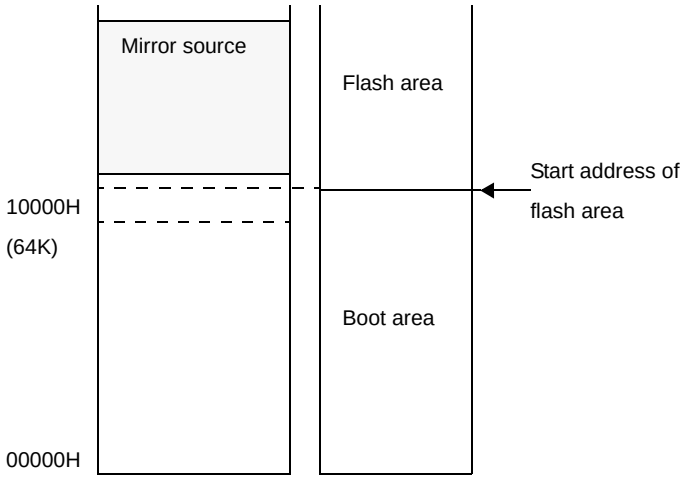


Figure 3-8. Memory Map Example 6



[Example]

To generate a branch table after the address 2000H and place the interrupt function:

<C source>

```
#pragma interrupt       INTP0   intp

void intp ( void ) {
}
```

(1) To place the interrupt function to the boot area (no -zf specified, -zt2000h specified)

<Output object of compiler>

```
                PUBLIC  _intp
                PUBLIC  _@vect06
@@BASE          CSEG    BASE
_intp :
                reti

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      _intp
```

Sets the first address of the interrupt function in the interrupt vector table.

(2) To place the interrupt function in the flash area (-zf specified, -zt2000h specified)

<Output object of compiler>

```
                PUBLIC  _intp
@ECODE          CSEG    BASE
_intp :
                reti

@EVECT06        CSEG    AT      0200CH
                br      !!_intp
```

Sets the first address of the interrupt function in the branch table.
The address value of the branch table is 2000H + 4 * (0006H / 2) since the first address of the branch table is 200CH and the interrupt vector address (2 bytes) is 0006H.
The interrupt vector library performs the setting of the address 200CH in the interrupt vector table.

<Library for interrupt vector 06>

```
                PUBLIC  _@vect06

@@VECT06        CSEG    AT      0006H
_@vect06 :
                DW      200CH
```

Function of function call from boot area to flash area (#pragma ext_func)

The #pragma instruction specifies the function name and ID value in the flash area called from the boot area. It becomes possible to call a function in the flash area from the boot area.

Caution This function enables the flash memory rewriting function of devices.

[Function]

- Function calls from the boot area to the flash area are executed via the flash area branch table.
- From the flash area, functions in the boot area can be called directly.

[Effect]

- It becomes possible to call a function in the flash area from the boot area.

[Usage]

- The following #pragma instruction specifies the function name and ID value in the flash area called from the boot area.

```
#pragma ext_func        function-name  ID-value
```

- This #pragma instruction is described at the beginning of the C source.
- The following items can be described before this #pragma instruction.
  - Comments
  - Instructions not to generate the definition/reference of variables or functions among the preprocess instructions.

[Restrictions]

- The ID value is set at 0 to 255 (0xFF).
- An error occurs if a file containing a #pragma ext_func is compiled without specifying the -zt option or the -zz option.
- For the same function with a different ID value and a different function with the same ID value, an error will occur.
  (1) and (2) below are errors.

(1) #pragma ext_func f1 3
#pragma ext_func f1 4

(2) #pragma ext_func f1 3
#pragma ext_func f2 3

- If a function is called from the boot area to the flash area and there is no corresponding function definition in the flash area, the linker cannot conduct a check. This is the user's responsibility.
- The callt functions can only be located in the boot area. If the callt functions are defined in the flash area (when the -zf option is specified), it results in an error.

- When the -rf option is specified for the small or medium model, and when the -rn option is specified for the large model, the suffix "_f" must be added to the library function name when calling a standard library function that takes a (const *) argument (warning W0072 is always output).
  The following standard library arguments take (const *) arguments.
  sprintf/sscanf/printf/scanf/vprintf/vsprintf/puts/atoi/atol/strtol/stltoul/atof/strtod/bsearch/qsort/memcpy/memmove/strcpy/strncpy/strcat/strncat/memcmp/strcmp/strncmp/memchr/strchr/strcspn/strpbrk/strrchr/strspn/strstr/strtok/strlen/strcoll/strxfrm

[Cautions]

- A program ceases to conform to the ANSI standard when the -rf option is specified in the small or medium model, and when the -rn option is specified for the large model. Warning W0073 is issued if the strict ANSI option -za is specified.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- If the #pragma ext_func is not used, no corrections are necessary.
- To perform the function call from the boot area to the flash area, make the change in accordance with USAGE above.

(2) From the 78K0R C compiler to another C compiler

- Delete the #pragma ext_func instruction or divide it by #ifdef.
- To perform the function call from the boot area to the flash area, the following change is required.

[Example]

- In the case that the branch table is generated after address 2000H and functions f1 and f2 in the flash area are called from the boot area.

<C source>

```
(1) Boot area side

#pragma interrupt INTP0 intf0
#pragma ext_func f1 3
#pragma ext_func f2 4

void f1 ( ), f2 ( );

void func ( ) {
        f1 ( );
        f2 ( );
}
```

```
(2) Flash area side

#pragma interrupt INTP1 intf1
#pragma ext_func f1 3
#pragma ext_func f2 4

void f1 ( ) {
}

void f2 ( ) {
}

void intf1 ( ) {
}
```

Remarks 1. #pragma ext_func f1 3 means that the branch destination to function f1 is located in starting address of the branch table + 4 * 64 + 4 * 3.

2. #pragma ext_func f2 4 means that the branch destination to function f2 is located in starting address of the branch table + 4 * 64 + 4 * 4.

3. 4 * 64 bytes from the beginning of the branch table are dedicated to interrupt functions (including the startup routine).

<Output object of compiler>

(1) When allocation address of flash area branch table is within 64 KB

```
(1) Boot area side (no specified -zf and specified -zt2000h)

@@CODEL         CSEG
_func :
        call    !0210CH
        call    !02110H
        ret
@@VECT08        CSEG    AT    0008H
_@vect08:
                DW      _intf0
```

```
(2) Flash area side (specified -zf)

@ECODEL         CSEG
_f1:
        ret
_f2:
        ret
_intf1:
        reti

@EVECT0A        CSEG    AT    02014H
        br      !!_intf1

@EXT03          CSEG    AT    0210CH
        br      !!_f1
        br      !!_f2
```

```
(3) Interrupt vector library for 0A

@@VECT0A        CSEG    AT    000AH
_@vect0a:
                DW      2014H
```

(2) When allocation address of flash area branch table is not within 64 KB
When flash area branch table starting address is 13000H

```
(1) Boot area side (no specified -zf and specified -zt2000h)

@@CODEL         CSEG
_func :
        call    !!01310CH
        call    !!013110H
        ret

@@VECT08        CSEG    AT    0008H
_@vect08:
                DW      _intf0
```

```
(2) Flash area side(specified -zf and specified -zt2000h

@ECODEL        CSEG
_f1:
        ret
_f2:
        ret
_intf1:
        reti

@EVECT0A        CSEG    AT    013014H
        br      !!_intf1

@EXT03          CSEG    AT    01310CH
        br      !!_f1
        br      !!_f2
```

```
(3) Interrupt vector library for 0A

@@BASE          CSEG    BASE
?@vect0a:       br      !!013014H

@@VECT0A        CSEG    AT    000AH
_@vect0a:
                DW      ?@vect0a
```

Mirror source area specification

The -mi0/-mi1 options instruct the compiler to generate code for a specified mirror source area.

[Function]

- When the -mi0 option is specified, code for 0 in MAA is generated.
- When the -mi1 option is specified, code for 1 in MAA is generated.
- A link error occurs when modules have been compiled with different -mi0/-mi1 option specifications.
- When the -mi option is not specified, code for 0 in MAA is generated.
- By default the linker's -mi option is set to the value of the compiling -mi option.
- Unless specified, the -mi option is set to 0.
- A link error occurs if the value of the linker -mi option is different from the value of the compiler -mi option.
- See the user's manual of the target device for more information about the mirror area and the MAA bit.

[Effect]

- The compiler generates code for the specified mirror source area.

[Usage]

- At compiling, specify the -mi0 or -mi1 option.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- The -mi option can be specified to select the mirror source area. No modifications to source files are required.

(2) From the 78K0R C compiler to another C compiler

- Source files can be compiled on other C compilers with no modifications required

Method of int expansion limitation of argument/return value (-zb)

The -zb option is specified during compiling, the object code is reduced and the execution speed improved.

[Function]

- When the type definition of the function return value is char/unsigned char, the int expansion code of the return value is not generated.
- When the prototype of the function argument is defined and the argument definition of the prototype is char/unsigned char, the int expansion code of the argument is not generated.

[Effect]

- The object code is reduced and the execution speed improved since the int expansion codes are not generated.

[Usage]

- The -zb option is specified during compiling.

[Example]

<C source>

```
unsigned char   func1 ( unsigned char x, unsigned char y );
unsigned char   c, d, e ;

void main ( void ) {
        c = func1 ( d, e );
        c = func2 ( d, e );
}

unsigned char   func1 ( unsigned char x, unsigned char y ) {
        return x + y ;
}
```

(1) When the -zb option is specified

<Output object of compiler>

```
_main :
; line  5 :             c = func1 ( d, e );
        mov     x, !_e
        push    ax
        mov     x, !_d                  ; Do not execute int expansion
        call    !_func1
        pop     ax
        mov     a, c
        mov     !_c, a
; line  6 :             c = func2 ( d, e );
        mov     x, !_e
        clrb    a                       ; Execute int expansion since there is no
                                        ; prototype declaration
        push    ax
        mov     x, !_d
        mov     x, #00H
        xch     a, x                    ; Execute int expansion since there is no
                                        ; prototype declaration
        call    !_func2
        pop     ax
        mov     a, c
        mov     !_c, a

; line 7 :      }
        ret
; line 8 :
; line 9 :      unsigned char func1 ( unsigned char x, unsigned char y ) {
_func1 :
        push    hl
        push    ax
        movw    ax, sp
        movw    hl, ax
        mov     a, [hl]
        mov     x, a
        mov     a, [hl + 6]
        movw    hl, ax
; line  10 :            return x + y ;
        mov     a, l
        add     a, h
        mov     c, a                  ; Do not execute int expansion
; line 11 :     }
        pop     ax
        pop     hl
        ret
```

[Restrictions]

- If the files are different between the definition of the function body and the prototype declaration to this function, the program may operate incorrectly.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- If the prototype declarations for all definitions of function bodies are not correctly performed, perform correct prototype declaration. Alternatively, do not specify the -zb option.

(2) From the 78K0R C compiler to another C compiler

- No modification is needed.

Memory manipulation function (#pragma inline)

An object file is generated by the output of the standard library functions memcpy and memset with direct inline expansion.

[Function]

- An object file is generated by the output of the standard library memory manipulation functions memcpy and memset with direct inline expansion instead of function call.
- When there is no #pragma directive, the code that calls the standard library functions is generated.

[Effect]

- Compared with when a standard library function is called, the execution speed is improved.
- Object code is reduced if a constant is specified for the specified character number.

[Usage]

- The function is described in the source in the same format as a function call.
- The following items can be described before #pragma inline.
  - Comments
  - Other #pragma directives
  - Preprocess directives that do not generate variable definitions/references or function definitions/references

[Example]

<C source>

```
#pragma inline

char    ary1[100], ary2[100] ;

void main ( void ) {
        memset ( ary1, 'A', 50 );
        memcpy ( ary1, ary2, 50 );
}
```

<Output object of compiler>

```
_main :
        push    hl
; line  5 :     memset ( ary1, 'A', 50 );
        movw    de, #loww ( _ary1 )
        mov     a, #041H        ; 65
        mov     c, #032H        ; 50
L0003 :
        mov     [de], a
        incw    de
        dec     c
        bnz     $L0003
; line  6 :     memcpy ( ary1, ary2, 50 );
        movw    de, #loww ( _ary1 )
        movw    hl, #loww ( _ary2 )
        mov     c, #032H        ; 50
L0005 :
        mov     a, [hl]
        mov     [de], a
        incw    de
        incw    hl
        dec     c
        bnz     $L0005
; line  7 : }
        pop     hl
        ret
```

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Modification is not needed if the memory manipulation function is not used.
- When changing the memory manipulation function, use the method above.

(2) From the 78K0R C compiler to another C compiler

- The #pragma inline directive should be deleted or delimited using #ifdef.

Absolute address allocation specification (__directmap)

Declare __directmap in the module in which the variable to be allocated in an absolute address is to be defined. Variables can be allocated to the arbitrary address.

[Function]

- The initial value of an external variable declared by __directmap and a static variable in a function is regarded as the allocation address specification, and variables are allocated to the specified addresses.
  Specify the allocation address using integers.
- The __directmap variable in the C source is treated as an static variable.
- Because the initial value is regarded as the allocation address specification, the initial value cannot be defined and remains an undefined value.
- The specifiable address specification range, secured area range linked by the module for securing the area for the specified addresses, and variable duplication check range are shown in the table below.

Item	Range	
	When Small Model or Medium Model Is Specified	When Large Model Is Specified
Address Specification Range	0xf0000 - 0xfffff	0x00000 - 0xfffff
Secured Area Range	0xffd00 - 0xffeff	0xffd00 - 0xffeff
Duplication Check Range	Start address - end address of device internal RAM	Start address - end address of device internal RAM

- If the address specification is outside the address specification range, an error is output.
- A variable that is declared with __directmap cannot be allocated to an area that extends over a boundary of the following areas. If allocated, an error will be output.
  - saddr area (0xffe20 to 0xffeff)
  - sfr area or an area with which saddr area overlaps (0xfff00 to to 0xfff1f)
  - sfr area (0xfff20 to 0xfffff)
  - 2nd sfr area (Varies depending on the device used.)
- If the allocation address of a variable declared by __directmap is duplicated and is within the duplication check range, a warning message (W0762) is output and the name of the duplicated variable is displayed.
- If the address specification range is inside the saddr area, the __sreg declaration is made automatically and the saddr instruction is generated.
- If char/unsigned char/short/unsigned short/int/unsigned int/long/unsigned long type variables declared by __directmap are bit referenced, sreg/__sreg must be specified along with __directmap. If they are not, an error will occur.
- If the specified address range is in the near area, the variable is regarded to be in the near area for accessing.
- If the specified address range is in neither the saddr area nor near area, the variable is regarded to be in the far area for accessing.
- If neither the __near nor __far type qualifier is specified, the variable is accessed in accordance with the memory model specifications.

- If a type qualifier is specified, the variable is accessed in accordance with the specification. If the specified address range and the type qualifier contradict, an error will be output.
  The table below lists the relationship between the address specification ranges, memory models, and type qualifiers.

Address Specification Range		Type Qualifier					
		__near __sreg	__far __sreg	__sreg	__near	__far	No Specification
In saddr area	Accessing method	sreg	sreg	sreg	sreg	sreg	sreg
	Pointer length	2 bytes	4 bytes	Small : 2 bytes Medium : 2 bytes Large : 4 bytes	2 bytes	4 bytes	Small : 2 bytes Medium : 2 bytes Large : 4 bytes
In near area	Accessing method	Error	Error	Error	near	far	Small : near Medium : near Large : far
	Pointer length				2 bytes	4 bytes	Small : 2 bytes Medium : 2 bytes Large : 4 bytes
In far area	Accessing method	Error	Error	Error	Error	far	Small : Error Medium : Error Large : far
	Pointer length					4 bytes	Small : Error Medium : Error Large : 4 bytes

[Effect]

- One or more variables can be allocated to the same arbitrary address.

[Usage]

- Declare __directmap in the module in which the variable to be allocated in an absolute address is to be defined.

```
__directmap               type-name   variable-name = allocation-address-specification ;
__directmap static        type-name   variable-name = allocation-address-specification ;
__directmap __sreg        type-name   variable-name = allocation-address-specification ;
__directmap __sreg static type-name   variable-name = allocation-address-specification ;
```

- If __directmap is declared for a structure/union/array, specify the address in braces {}.

[Example]

<C source>

```
__directmap     char    c = 0xffe00 ;
__directmap     __sreg  char    d = 0xffe20 ;
__directmap     __sreg  char    e = 0xffe21 ;
__directmap     struct  x {
        char    a ;
        char    b ;
} xx = { 0xffe30 };

void main ( void ) {
        c = 1 ;
        d = 0x12 ;
        e.5 = 1 ;
        xx.a = 5 ;
        xx.b = 10 ;
}
```

<Output object of compiler>

```
        PUBLIC  _main
_c      EQU     0FFE00H                 ; Addresses for variables declared by __directmap
_d      EQU     0FFE20H                 ; are defined by EQU
_e      EQU     0FFE21H                 ;
_xx     EQU     0FFE30H                 ;
        EXTRN   __mmfe00                ; For linking secured area modules
        EXTRN   __mmfe20                ; EXTRN output
        EXTRN   __mmfe21                ;
        EXTRN   __mmfe30                ;
        EXTRN   __mmfe31                ;
@@CODEL CSEG
_main :
; line 10 :
        oneb    !loww ( _c )
; line 11 :
        mov     _d, #012H               ; saddr instruction output because address
; line 12 :                             ; specified in saddr area
        set1    _e.5                    ; Bit manipulation possible because __sreg also used
; line 13 :
        mov     _xx, #05H               ; saddr instruction output because address
                                        ; specified in saddr area
; line 14 :
        mov     _xx + 1, #0AH           ; saddr instruction output because address
                                        ; specified in saddr area
; line 15 :
        ret
        END
```

[Restrictions]

- __directmap cannot be specified for function arguments, return values, or automatic variables. If it is specified in these cases, an error will occur.
- If an address outside the secured area range is specified, the variable area will not be secured, making it necessary to either describe a directive file or create a separate module for securing the area.
- The __directmap variable cannot be declared with extern because it is handled in the same way as the static variables.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- No modification is necessary if the keyword __directmap is not used.
- To change to the __directmap variable, modify according to the description method above.

(2) From the 78K0R C compiler to another C compiler

- Compatibility can be attained using #define (see "3.3.5 C source modifications for details).
- When the __directmap is being used as the absolute address allocation specification, modify according to the specifications of each compiler.

near/far area specification

An allocating place of the function and a variable can be designated specifically by adding the __near or __far type qualifier when a function or variable declared.

[Function]

- The location of a function or variable is specified explicitly by specifying a __near or __far type qualifier.

Qualifier	Location
__near type qualifier	near area (data:0F0000H to 0FFFFFH, code:000000H to 00FFFFH)
__far type qualifier	far area (000000H to 0FFFFFH)

- The pointer to the near area should be 2 bytes long, and that to the far area should be 4 bytes long.
- An error will occur if __near and __far type qualifiers are used together in declaration of the same variable or function.
- The __near and __far type qualifiers are handled as type qualifiers, grammatically.
- If specified together with __callt, __interrupt, __rtos_interrupt, __interrupt_brk, __sreg, or __boolean, the __near or __far type qualifier is ignored.
- An error will occur if __near and __far type qualifiers are specified together.
- If specified for an automatic variable, argument or register variable, the __near or __far type qualifier is ignored.
- Variables in the near area are accessed without using the ES register.
  The pointer length should be 2 bytes long.
- Variables in the far area are accessed by setting the ES register.
  The pointer length should be 4 bytes long.
- Functions in the near area are called with !addr16, and functions in the far area are called with !!addr20.
- Since there are no instructions that can call function pointers without referencing the CS register, be sure to set the CS register to call function pointers.
- Function pointers for functions in the near area output the code to set the CS register to 0.
- The highest byte of a far pointer is always undefined.
- Conversion from the near pointer or int to the far pointer, and from the near pointer to long results in the following operations.
  - "0xf" is added to the higher bytes of the variable pointer (0 is exceptional and zero-extended).
  - The function pointer is zero-extended.
- Addition and subtraction with the far pointer uses the lower 2 bytes, and the higher bytes do not change.
- ptrdiff_t is always int type.
- An equality operation with the far pointer uses the lower 3 bytes.
- A relational operation with the far pointer uses the lower 2 bytes. To compare pointers that do not point to the same object, the pointer must be converted to unsigned long. If the -za option is specified, the lower 3 bytes are used for comparison.
- The character string constants are allocated to the far area or near area, according to the memory model specified.

Memory Model	Location
Small model	near area
Medium model	near area
Large model	far area

- When the large model is used, pointers to automatic variables, arguments, and sreg variables are 4 bytes long.
- The following error checking is performed to detect cases in which the same variable or function is declared __near in the defining module and __far in another module, or the reverse. (See "Coding examples 2" below.)
  - A link error occurs when a variable or function is referenced if 1) it has been declared __near in the defining module, and 2) it is declared __far in the module where it is referenced.
  - Error checks are performed for up to 8 of any combination of pointer, array, or function declarator.
  - Error checks are performed only when the -g option is specified.

[Effect]

- Specification of the __far type qualifier enables functions and variables to be allocated to the far area and to be referenced.
- Specification of the __near type qualifier enables functions and variables to be allocated to the near area and to be referenced.
  The functions and variables allocated to the near area can be called or referenced with a short instruction.

[Usage]

- The __near or __far type qualifier is added to a function or variable declared.

[Example]

(1) Coding examples 1

```
__near int i1;
__far  int i2;
__far  int * __near p1;
__far  int * __near * __far p2;
__far  int func1();
__far  int * __near func2();
__near int ( * __far fp1 )();
__far  int * __near ( * __near fp2 )();
__near int * __far ( * __near fp3 )();
__near int * __near ( * __far fp4 )();
```

- i1 is int type and allocated to the near area.
- i2 is int type and allocated to the far area.
- p1 is a 4-byte type variable that points to "an int type in the far area". The variable itself is allocated to the near area.
- p2 is a 2-byte variable that points to a 4-byte type in the near area, which points to "an int type in the far area". The variable itself is allocated to the far area.
- func1 is a function that returns "an int type". The function itself is allocated to the far area.
- func2 is a function that returns a 4-byte type that points to "an int type in the far area". The function itself is allocated to the near area.
- fp1 is a 2-byte type variable that points to "a function in the near area, which returns an int type". The variable itself is allocated to the far area.
- fp2 is a 2-byte type variable that points to a function in the near area, which returns a 4-byte type that points
- to "an int type in the far area". The variable itself is allocated to the near area.
- fp3 is a 4-byte type variable that points to a function in the far area, which returns a 2-byte type that points to "an int type in the near area". The variable itself is allocated to the near area.
- fp4 is a 2-byte type variable that points to a function in the near area, which returns a 2-byte type that points to "an int type in the near area". The variable itself is allocated to the far area.

(2) Coding examples 2

- The following examples explain the error checking that is performed to detect cases in which the same variable or function is declared near in the defining module and far in another module, or the reverse.

```
(1)a.c

/* Definitions */
int     __near i1;
int     __far  i2;
int     __near * __near nnp1;
int     __near * __near nnp2;
int     __near * __far  fnp1;
int     __near * __near nnp3;
int     __far  * __near nfp1;
int     __far  * __near nffunc1(){}
int     __far  * __near nffunc2(){}
int     __far  * __far  fffunc1(){}
int     __near * __far  fnfunc1(){}
int     __far  * __far  fffunc2(){}
```

```
(2)b.c

/* extern  declarations */
extern int     __far  i1;
extern int     __near i2;
extern int     __near * __near nnp1;
extern int     __near * __far  nnp2;
extern int     __near * __near fnp1;
extern int     __far  * __near nnp3;
extern int     __near * __near nfp1;
extern int     __far  * __near nffunc1();
extern int     __far  * __far  nffunc2();
extern int     __far  * __near fffunc1();
extern int     __far  * __far  fnfunc1();
extern int     __near * __far  fffunc2();

void main(void) {
        i1 = 1;           /* OK */
        i2 = 1;           /* Error */
        *nnp1 = 1;        /* OK */
        *nnp2 = 1;        /* OK */
        *fnp1 = 1;        /* Error */
        *nnp3 = 1;        /* Error */
        *nfp1 = 1;        /* Error */
        nffunc1 ( );      /* OK */
        nffunc2 ( );      /* OK */
        fffunc1 ( );      /* Error */
        fnfunc1 ( );      /* Error */
        fffunc2 ( );      /* Error */
}
```

[Restrictions]

- Even if the __far type qualifier is specified, data cannot be allocated to an area extending over a 64 KB boundary.
  Functions can be allocated to an area extending over a 64 KB boundary.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- It is not necessary to modify the code if reserved word __near or __far is not used.

(2) From the 78K0R C compiler to another C compiler

- It is not necessary to modify the code if the __near or __far type qualifier is not used.
- If the __near or __far type qualifier is used, #define can be used for near/far area specification.

[Cautions]

- If the lower 2 bytes are used for a relational operation, data cannot be allocated to the last byte of a 64 KB boundary area. If allocated, an error will be output by the linker or compiler.
  This is because, ANSI-compliant operation[Note] is performed for the relational operation that uses the pointer that points to the range outside an array.

Note Constraints on relational operators prescribed by ANSI
If expression P points to an element of an array object and expression Q points to the last element of that array object, pointer expression Q+1 is larger than expression P.

- The size of the pointer for the far area is 4 bytes but the calculation object is the lower 3 bytes, so the highest byte is always undefined.

<Example>

```
union   tag {
        __far unsigned short    *ptr;
        unsigned long           ldata;
} un;
```

A value is written to un.ptr and then un.ldata is referenced; the highest byte then becomes undefined. To guarantee that the highest byte of un.ldata is 0, union un must first be initialized with 0.

- The linker checks the data location of sections with the following combination of segment type and relocation attribute.
  DSEG UNIT64KP
  DSEG PAGE64KP
  CSEG PAGE64KP
- If one of the above relocation attributes is changed using the #pragma section or link directive file, the linker does not check it.

Memory model specification

An allocating place of the function and a variable can be specifying by a memory model by specifying the -ms, -mm, or -ml option when compiling.

[Function]

- The location of a function or variable is specified.

Memory Model	Data	Function
Small model	near area	near area
Medium model	near area	far area
Large model	far area	far area

- If the __near or __far type qualifier is specified, the specified __near or __far type qualifier takes precedence.
- Small model
  Consists of a data portion of 64 KB and a code portion of 64 KB; 128 KB in total.
  The data ROM is allocated at 0000H to 0FFFFH or 10000H to 1FFFFH, and mirrored in FxxxxH.
  Codes are allocated at 00000H to 0FFFFH.
  Since the CS register value may be changed by specifying the __far type qualifier, be sure to set the CS register when calling a function pointer.
- Medium model
  Variables are allocated to the near area, and functions are allocated to the far area. Consists of a data portion of 64 KB and a code portion of 1 MB.
  The data ROM is allocated at 000000H to 00FFFFH or 010000H to 01FFFFH, and mirrored in FxxxxH. There are no limitations on locating codes.
- Large model
  Variables and functions are allocated to the far area. Consists of a data portion of 1 MB and a code portion of 1 MB. There are no limitations on locating data and codes.

[Usage]

- Specify the -ms, -mm, or -ml option during compilation.

Option	Explaination
-ms	Small model
-mm	Medium model
-ml	Large model

[Example]

<C source>

```
int     i ;
int     *p ;
void func( void ) { }
void ( *fp )( void );

void main( void ) {
        int     r;

        r = i ;         /* Data access */
        func();         /* Function call */
        r = *p ;        /* Data pointer */
        fp();           /* Function pointer */
}
```

<Output object of compiler>

(1) When small model is used

```
movw    hl, !_i
call    !_func
movw    de, !_p
movw    ax, [de]
movw    hl, ax
movw    ax, !_fp
mov     CS, #00H        ; 0
call    ax
```

(2) When medium model is used

```
movw    hl, !_i
call    !!_func
movw    de, !_p
movw    ax, [de]
movw    hl, ax
mov     a, !_fp + 2
mov     CS, a
movw    ax, !_fp
call    ax
```

(3) When large model is used>

```
mov     ES, #highw ( _i )
movw    hl, ES:!_i
call    !!_func
mov     ES, #highw ( _p )
mov     a, ES:!_p + 2
movw    de, ES:!_p
mov     ES, a
movw    ax, ES:[de]
movw    hl, ax
mov     ES, #highw ( _fp )
mov     a, ES:!_fp + 2
mov     CS, a
movw    ax, ES:!_fp
call    ax
```

[Restrictions]

- Even if the large model is specified, data cannot be allocated to an area that extends over 64 KB boundaries.
- Modules for which a different memory model is specified cannot be linked.
- The size of variables with/without initial values allocated to the far area are (64K - 1) bytes each, per load module file (Note: 64KB if the -za option is specified).
  This size can be increased by changing the section name that includes variables with/without initial values in a certain file to another output section name, using the function of "Changing compiler output section name (#pragma section ...)".
  In this case, the startup routine and termination routine must be modified (refer to [Examples of Changing startup Routine in Connection with Changing Section Name Related to ROMization] in "Changing compiler output section name (#pragma section ...)").
- The maximum size per output section name does not change.
- If the -za option is not specified, data cannot be allocated to the last byte of a 64 KB boundary area (refer to CAUTIONS in "near/far area specification").

Allocating ROM data specification

An allocating place of the ROM data can be designated specifically near or far area.

[Function]

- The -rf option places ROM data in a far area.
- The -rn option places ROM data in a near area.
- When neither the -rf nor the -rn option is specified, the placement of ROM data depends on the memory model.
- The placement of ROM data is determined by the following specifications, listed in order of priority from highest priority to lowest.

(1) near or far specification by specification of the start address of the flash area and the address of the mirror source area (see "Flash area branch table and flash area allocation (#pragma ext_table)").

(2) __near or __far keyword

(3) -rn or -rf option specification

(4) Memory model

- ROM data refers to the following types of data.
  - Variables declared as const
  - String literals
  - Initial values of auto aggregate type variables (arrays and structures)
  - Switch statement branch tables

[Effect]

- It's possible to allocate ROM data in any area far or near area.

[Usage]

- Specify the -rf or -rn option at compiling.

[Restrictions]

- When the same const variable is referenced by different modules, it is placed according to the ROM data specification priorities listed above, and an error check is performed. See "near/far area specification" about the error check.

[Compatibility]

(1) From another C compiler to the 78K0R C compiler

- Specify the placement of ROM data by recompiling with the -rf or -rn option specified. There is no need to modify the source program.

(2) From the 78K0R C compiler to another C compiler

- Compile the source program on other C compilers with no modifications.

3.3.5 C source modifications

The compiler generates efficient object code when using the extended functions. But these functions are designed for use on 78K0R microcontrollers. If programs make use of the extended functions, they must be modified when porting them for use on other devices.

This section explains techniques that can use to port programs from other C compilers to 78K0R C compiler, and from 78K0R C compiler to other C compilers.

(1) From another C compiler to the 78K0R C compiler

- #pragma[Note]
  If the other C compiler supports the #pragma directive, the C source must be modified. The method and extent of modifications to the C source depend on the specifications of the other C compiler.
- Extended specifications
  If the other C compiler has extended specifications such as addition of keywords, the C source must be modified. The method and extent of modifications to the C source depend on the specifications of the other C compiler.

Note #pragma is one of the preprocessing directives supported by ANSI. The character string following the #pragma is identified as a directive to the compiler. If the compiler does not support this directive, the #pragma directive is ignored and the compile will be continued until it properly ends.

(2) From the 78K0R C compiler to another C compiler

- Because the 78K0R C compiler has added keywords as the extended functions, the C source must be made portable to the other C compiler by deleting such keywords or invalidating them with #ifdef.

Following are some examples.

(a) To invalidate a keyword (Same applies to callf, sreg, and norec, etc.)

```
#ifndef __K0R__
#define callt           /* Makes callt as ordinary function */
#endif
```

(b) To change from one type to another

```
#ifndef __K0R__
#define bit     char    /* Changes bit type to char type variable*/
#endif
```

3.4 Function Call Interface

This section explains the following features of the function call interface.

- Return values (common to all functions)
- Ordinary function call interface

3.4.1 Return values

- The return value of a function is stored in registers or carry flags.
- The locations at which a return value is stored are listed below.s

Table 3-16. Storage Locations of Return Values

Type	Storage Location
1 bit	CY
1-byte or 2-byte integer	BC
near pointer	BC
4-byte integer	BC (lower), DE (upper)
far pointer	BC (lower), DE (upper)
Floating-point number	BC (lower), DE (upper)
Structure	The structure to be returned is copied into private storage for the function, and the address of the copy is stored in BC and DE.

3.4.2 Ordinary function call interface

(1) Passing arguments

- The second and following arguments are passed to functions on the stack.
- The first argument is passed to the function definition side via a register or stack.
  The location where the first argument is passed is shown in the table below.

Table 3-17. Location Where First Argument Is Passed (Function Calling Side)

Type	Storage Location
1-byte data[Note] 2-byte data[Note]	AX
Pointer to near data	AX
3-byte data[Note] 4-byte data[Note]	AX, BC
Pointer to function, Pointer to far data	AX, BC
Floating-point number	AX, BC
Other	On the stack

Note 1-byte to 4-byte data includes structures, unions, and pointers.

(2) Storage locations of arguments and auto variables

- An argument or automatic variable is assigned to a register at the top of the function, by declaring the argument or automatic variable with register or specifying the -qv option. Other arguments and automatic variables are stored in a stack.
- If an argument, which is passed from the function call side via a stack, is not assigned to registers, the location for passing is the location to be assigned.
- Arguments and automatic variables are assigned to register HL, unless otherwise there are no stack frames.
  Arguments and automatic variables can also be assigned to _@KREGxx if the -qr option is specified. Refer to "3.5 List of saddr Area Labels" for _@KREGxx.
- Arguments and automatic variables are assigned to registers in the order of reference frequency.
  Arguments and automatic variables that are rarely referenced may not be assigned to registers, even if the argument or automatic variable is declared with register or the -qv option is specified.
- The registers to which arguments or automatic variables are assigned are saved and restored by the function definition side.

(3) Examples

(a) Examples 1

<C source>

```
void func0 ( register int, int );

void main ( void ) {
        func0 ( 0x1234, 0x5678 );
}

void func0 ( register int p1, int p2 ) {
        register int    r ;
        int     a ;
        r = p2 ;
        a = p1 ;
}
```

<1> -When -qr option is specified

<Assembly source code generated by compiler>

```
_main :
; line  4 :     func0 ( 0x1234, 0x5678 );
        movw    ax,  #05678H    ; 22136
        push    ax                      ; 2nd and following arguments passed on
                                        ; stack
        movw    ax, #01234H     ; 4660  ; 1st argument passed in register
        call    !!_func0                ; Function call
        pop     ax                      ; Release stack used for function call
; line  5 : }
        ret
; line  6 :
; line  7 : void func0 ( register int p1, int p2 ) {
_func0 :
        push    hl
        movw    de, _@KREG14
        push    de                      ; Save saddr area for register variable
        movw    de, _@KREG12
        push    de                      ; Save saddr area for register variable
        movw    _@KREG14, ax            ; Assign 1st argument p1 to saddr
        push    ax                      ; Reserve storage for auto variable a
        movw    hl, sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax, [hl+12]     ; p2    ; Argument p2
        movw    _@KREG12, ax    ; r     ; Auto variable r
; line 11 :     a = p1 ;
        movw    ax, _@KREG14    ; p1    ; Argument p1
        movw    [hl], ax        ; a     ; Auto variable a
; line 12 : }
        pop     ax                      ; Release storage for auto variable a
        pop     ax
        movw    _@KREG12, ax            ; Restores saddr area for register
                                        ; argument
        pop     ax
        movw    _@KREG14, ax            ; Restores saddr area for register
                                        ; argument
        pop     hl
        ret
```

(b) Example 2

<C source>

```
void func1 ( int, register int );

void main ( void ) {
        func1 ( 0x1234, 0x5678 );
}

void func1 ( int p1, register int p2 ) {
        register int    r ;
        int     a ;
        r = p2 ;
        a = p1 ;
}
```

<1> When -qr option is specified

<Assembly source code generated by compiler>

```
_main :
; line  4 :     func1 ( 0x1234, 0x5678 );
        movw    ax, #05678H     ; 22136
        push    ax                      ; 2nd and following arguments passed on
                                        ; stack
        movw    ax, #01234H     ; 4660  ; 1st argument passed in register
        call    !!_func1                ; Function call
        pop     ax                      ; Release stack used for function call
; line  5 : }
        ret
; line  6 :
; line  7 : void func1 ( int p1, register int p2 ) {

_func0 :
        push    hl
        push    ax                      ; Place 1st argument p1 on stack
        movw    de, _@KREG14
        push    de                      ; Save saddr area for register variable
        movw    de, _@KREG12
        push    de                      ; Save saddr area for register variable
        movw    ax, [sp+12]
        movw    _@KREG12, ax            ; Assign argument p2 to saddr
        push    ax                      ; Reserve storage for auto variable a
        movw    hl, sp
; line  8 :     register int    r ;
; line  9 :     int             a ;
; line 10 :     r = p2 ;
        movw    ax, _@KREG12    ; p2    ; Argument p2
        movw    _@KREG14, ax    ; r     ; Auto variable r
; line 11 :     a = p1 ;
```

```
        movw    ax, [hl+6]      ; p1    ; Argument p1
        movw    [hl], ax        ; a     ; Auto variable a
; line 12 : }
        pop     ax                      ; Release storage for auto variable a
        pop     ax
        movw    _@KREG12, ax            ; Restores saddr area for register
                                        ; variable
        pop     ax
        movw    _@KREG14, ax            ; Restores saddr area for register
                                        ; variable
        pop     ax                      ; Release storage for 1st argument p1
        pop     hl
        ret
```

3.5 List of saddr Area Labels

78K0R C compiler uses the following labels to reference addresses in the saddr area. Therefore, the names in the following tables cannot be used in C and assembler source programs.

(1) Register variables

Label Name	Address
_@KREG00	0FFEB4H
_@KREG01	0FFEB5H
_@KREG02	0FFEB6H
_@KREG03	0FFEB7H
_@KREG04	0FFEB8H
_@KREG05	0FFEB9H
_@KREG06	0FFEBAH
_@KREG07	0FFEBBH
_@KREG08	0FFEBCH
_@KREG09	0FFEBDH
_@KREG10	0FFEBEH
_@KREG11	0FFEBFH
_@KREG12	0FFEC0H[Note]
_@KREG13	0FFEC1H[Note]
_@KREG14	0FFEC2H[Note]
_@KREG15	0FFEC3H[Note]

Note When the arguments of the function are declared by register or the -qv option is specified and the -qr option is specified, arguments are allocated to the saddr area.

(2) For Works

Label Name	Address
_@NRARG0	0FFEC4H
_@NRARG1	0FFEC6H
_@NRARG2	0FFEC8H
_@NRARG3	0FFECAH
_@NRAT00	0FFECCH
_@NRAT01	0FFECDH
_@NRAT02	0FFECEH
_@NRAT03	0FFECFH
_@NRAT04	0FFED0H
_@NRAT05	0FFED1H
_@NRAT06	0FFED2H
_@NRAT07	0FFED3H

(3) For Segment information

Label Name	Address
_@SEGAX	0FFED4H
_@SEGBC	0FFED5H
_@SEGDE	0FFED6H
_@SEGHL	0FFED7H

(4) Runtime library arguments

Label Name	Address
_@RTARG0	0FFED8H
_@RTARG1	0FFED9H
_@RTARG2	0FFEDAH
_@RTARG3	0FFEDBH
_@RTARG4	0FFEDCH
_@RTARG5	0FFEDDH
_@RTARG6	0FFEDEH
_@RTARG7	0FFEDFH

3.6 List of Segment Names

This section explains all the segments that the compiler outputs and their locations.

The tables below list the relocation attributes that appear in the tables of this section.

- CSEG relocation attributes

CALLT0	Allocates the specified segment so that the start address is a multiple of two within the range of 80H to BFH.
AT absolute expression	Allocates the specified segment to an absolute address (within the range of 00000H to FFEFFH).
UNITP	Allocates the specified segment so that the start address is a multiple of two within any position (within the range of C0H to EFFFEH).

- DSEG relocation attributes

SADDRP	Allocates the specified segment so that the start address is a multiple of two within the range of FFE20H to FFEFFH in the saddr area.
UNITP	Allocates the specified segment so that the start address is a multiple of two within any position (default is within the RAM area).

3.6.1 List of segment names

(1) Program areas and data areas

Section Name	Segment Type	Relocation Attribute	Description
@@CODE	CSEG	BASE	Segment for code portion (allocated to near area)
@@CODEL	CSEG		Segment for code portion (allocated to far area)
@@LCODE	CSEG	BASE	Segment for library code (allocated to near area)
@@LCODEL	CSEG		Segment for library code (allocated to far area)
@@CNST	CSEG	MIRRORP	ROM data[Note 1]
@@CNSTL	CSEG	PAGE64KP	ROM data[Note 1]
@@R_INIT	CSEG	UNIT64KP	Segment for near initialized data (with initial value)
@@RLINIT	CSEG	UNIT64KP	Segment for far initialized data (with initial value)
@@R_INIS	CSEG	UNIT64KP	Segment for initialized data (sreg variable with initial value)
@@CALT	CSEG	CALLT0	Segment for callt function table
@@VECT*nn*	CSEG	AT 00*mm*H	Segment for vector table[Note 2]
@@BASE	CSEG	BASE	Segment for callt function and interrupt function
@@LBASE	CSEG	BASE	Segment for library and callt function
@@INIT	DSEG	BASEP	Segment for data area (with initial value, allocated to near area)
@@INITL	DSEG	UNIT64KP	Segment for data area (with initial value, allocated to far area)
@@DATA	DSEG	BASEP	Segment for data area (without initial value, allocated to near area)
@@DATAL	DSEG	UNIT64KP	Segment for data area (without initial value, allocated to far area)
@@INIS	DSEG	SADDRP	Segment for data area (sreg variable with initial value)
@@DATS	DSEG	SADDRP	Segment for data area (sreg variable without initial value)

Section Name	Segment Type	Relocation Attribute	Description
@@BITS	BSEG		Segment for boolean type and bit type variables

Notes 1. ROM data refers to the following types of data.
- Segment for const variables
- Table reference for switch-case statement
- Unknown character-string constant
- Data of initial value of an auto variable

2. The value of *nn* and *mm* changes depending on the interrupt types.

(2) Flash memory areas

Section Name	Segment Type	Relocation Attribute	Description
@ECODE	CSEG	BASE	Segment for code portion (allocated to near area)
@ECODEL	CSEG		Segment for code portion (allocated to far area)
@LECODE	CSEG	BASE	Segment for library code (allocated to near area)
@LECODEL	CSEG		Segment for library code (allocated to far area)
@ECNST	CSEG	MIRRORP	ROM data[Note 1]
@ECNSTL	CSEG	PAGE64KP	ROM data[Note 1]
@ER_INIT	CSEG	UNIT64KP	Segment for near initialized data (with initial value)
@ERLINIT	CSEG	UNIT64KP	Segment for far initialized data (with initial value)
@ER_INIS	CSEG	UNIT64KP	Segment for initialized data (sreg variable with initial value)
@EVECT*nn*	CSEG	AT *mmmm*H	Segment for vector table[Note 2]
@EXT*xx*	CSEG	AT *yyyy*H	Segment for flash area branch table[Note 3]
@EINIT	DSEG	BASEP	Segment for data area (with initial value, allocated to near area)
@EINITL	DSEG	UNIT64KP	Segment for data area (with initial value, allocated to far area)
@EDATA	DSEG	BASEP	Segment for data area (without initial value, allocated to near area)
@EDATAL	DSEG	UNIT64KP	Segment for data area (without initial value, allocated to far area)
@EINIS	DSEG	SADDRP	Segment for data area (sreg variable with initial value)
@EDATS	DSEG	SADDRP	Segment for data area (sreg variable without initial value)
@EBITS	BSEG		Segment for boolean type and bit type variables
@ECALT	CSEG		Dummy segment

Notes 1. ROM data refers to the following types of data.
- Segment for const variables
- Table reference for switch-case statement
- Unknown character-string constant
- Data of initial value of an auto variable

2. The value of *nn* and *mmmm* changes depending on the interrupt types.

3. The values of *xx* and *yyyy* changes depending on the ID of the flash area function.

3.6.2 Segment allocation

Segment Type	Location (Default)
CSEG	ROM
BSEG	saddr area of RAM
DSEG	RAM

3.6.3 C source example

```
#pragma INTERRUPT       INTP0   inter   rb1     /* Interrupt vector*/

void main ( void );                             /* Function prototype declaration */
const   int     i_cnst = 1 ;                    /* const variable*/
callt   void f_clt ( void );                    /* callt function prototype declaration*/
boolean b_bit ;                                 /* boolean type variable*/
long    l_init = 2 ;                            /* Initialized external variable*/
int     i_data ;                                /* Uninitialized external variable*/
s
__sreg  int     sr_inis = 3 ;                   /* Iinitialized sreg variables*/
__sreg  int     sr_dats ;                       /* Uninitialized sreg variable*/

void main ( void ) {                            /* Function definition*/
        int     i ;
        i = 100 ;
}

void inter ( void ) {                           /* Interrupt function definition*/
}

callt   void f_clt ( void ) {                   /* callt function definition*/
}
```

3.6.4 Example of output assembler module

Directives and instructions sets in assembly language source output vary according to the target device.
For details, see "CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICTION".

(1) Small model

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cxxx sample.c -ms -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
```

```
$KANJICODE  SJIS
$TOL_INF    03FH, 0100H, 00H, 00H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; Segment for booleantype and bittype variable
_b_bit      DBIT

@@CNST      CSEG    MIRRORP             ; Segment for const variable
_i_cnst :   DW      01H             ; 1

@@R_INIT    CSEG    UNIT64KP            ; Segment for initialization data (Initialized
                                        ; external variable)
            DW      00002H, 00000H  ; 2

@@INIT      DSEG    BASEP              ; Segment for tentative data (with initial value)
_l_init :   DS      ( 4 )

@@DATA      DSEG    BASEP               ; Segment for tentative data (without initial
                                        ; value)
_i_data :   DS      ( 2 )

@@R_INIS    CSEG    UNIT64KP            ; Segment for initialization data (Iinitialized
                                        ; sreg variable)
            DW      03H             ; 3

@@INIS      DSEG    SADDRP              ; Segment for tentative data area (Iinitialized
                                        ; sreg variable)
_sr_inis :  DS      ( 2 )

@@DATS      DSEG    SADDRP             ; Segment for tentative data area (Uninitialized
                                        ; sreg variable)
_sr_dats :  DS      ( 2 )

@@CALT      CSEG    CALLT0              ; Segment for callt function table
?f_clt :    DW      _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* Interrupt function */
; line  2 :
```

```
; line  3 : void main ( void );            /* Function prototype declaration */
; line  4 : const   int     i_cnst = 1 ;    /* const variable */
; line  5 : callt   void f_clt ( void );    /* calltFunction prototype declaration */
; line  6 : boolean b_bit ;                 /* boolean type variable */
; line  7 : long    l_init = 2 ;            /* Initialized external variable */
; line  8 : int     i_data ;                /* Uninitialized external variable */
; line  9 : sreg    int     sr_inis = 3 ;   /* Iinitialized sreg variable */
; line 10 : sreg    int     sr_dats ;       /* Uninitialized sreg variable */
; line 11 :
; line 12 : void main ( ) {                 /* Function definition */

@@CODE       CSEG  BASE                ; Segment for code portion
_main :
        push    hl
; line 13 :         int     i ;
; line 14 :         i = 100 ;
        movw    hl, #064H         ; 100
; line 15 : }
        pop   hl
        ret
; line 16 :
; line 17 : void inter ( ) {                /* Interrupt function definition */

@@BASE          CSEG    BASE            ; Segment for callt and interrupt function
_inter :
; line 18 : }
         reti
; line 19 :
; line 20 : callt   void f_clt ( ) {        /* callt function definition */
_f_clt:
; line 21 : }
         ret
@@VECT06        CSEG    AT      0006H   ; Segment for vector table
_@vect06 :
                DW      _inter
                END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

(2) Medium model

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cxxx sample.c -mm -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH, 0100H, 00H, 04000H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; Segment for booleantype and bittype variable
_b_bit      DBIT

@@CNST      CSEG    MIRRORP             ; Segment for const variable
_i_cnst :   DW      01H                 ; 1

@@R_INIT    CSEG    UNIT64KP            ; Segment for initialization data (Initialized
                                        ; external variable)
            DW      00002H, 00000H  ; 2

@@INIT      DSEG    BASEP              ; Segment for tentative data (with initial value)
_l_init :   DS      ( 4 )

@@DATA      DSEG    BASEP               ; Segment for tentative data (without initial
value)
_i_data :   DS      ( 2 )

@@R_INIS    CSEG    UNIT64KP            ; Segment for initialization data (Iinitialized
                                        ; sreg variable)
            DW      03H             ; 3
```

```
@@INIS      DSEG    SADDRP              ; Segment for tentative data area (Iinitialized
                                        ; sreg variable)
_sr_inis :  DS      ( 2 )

@@DATS      DSEG    SADDRP              ; Segment for tentative data area (Uninitialized
                                        ; sreg variable)
_sr_dats :  DS      ( 2 )

@@CALT      CSEG    CALLT0              ; Segment for callt function table
?f_clt :    DW      _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* Interrupt function*/
; line  2 :
; line  3 : void main ( void );              /* Function prototype declaration */
; line  4 : const   int     i_cnst = 1 ;     /* const variable */
; line  5 : callt   void f_clt ( void );     /* calltFunction prototype declaration */
; line  6 : boolean b_bit ;                  /* boolean type variable */
; line  7 : long    l_init = 2 ;             /* Initialized external variable */
; line  8 : int     i_data ;                 /* Uninitialized external variable */
; line  9 : __sreg  int     sr_inis = 3 ;    /* Iinitialized sreg variable */
; line 10 : __sreg  int     sr_dats ;        /* Uninitialized sreg variable */
; line 11 :
; line 12 : void main ( ) {                  /* Function definition */

@@CODEL     CSEG                        ; Segment for code portion
_main :
         push    hl
; line 13 :          int     i ;
; line 14 :          i = 100 ;
         movw    hl, #064H           ; 100
; line 15 : }
         pop     hl
         ret
; line 16 :
; line 17 : void inter ( ) {                 /* Interrupt function definition */

@@BASE   CSEG    BASE                   ; Segment for callt and interrupt function
_inter :
; line 18 : }
         reti
; line 19 :
; line 20 : callt   void f_clt ( ) {         /* callt function definition*/
_f_clt:
; line 24 : }
         ret
@@VECT06 CSEG    AT      0006H          ; Segment for vector table
_@vect06 :
         DW      _inter
         END
```

```
; Target chip : uPDxxxx
; Device file : xx.xxx
```

(3) Large model

```
; 78K0R C Compiler Vx.xx Assembler Source       Date:xx xxx xxxx Time:xx:xx:xx

; Command   : -cxxx sample.c -ml -sa -ng
; In-file   : sample.c
; Asm-file  : sample.asm
; Para-file :

$PROCESSOR ( xxx )
$NODEBUG
$NODEBUGA
$KANJICODE  SJIS
$TOL_INF    03FH, 0100H, 00H, 08000H, 00H, 00H, 00H

            PUBLIC  _inter
            PUBLIC  _main
            PUBLIC  _i_cnst
            PUBLIC  ?f_clt
            PUBLIC  _b_bit
            PUBLIC  _l_init
            PUBLIC  _i_data
            PUBLIC  _sr_inis
            PUBLIC  _sr_dats
            PUBLIC  _f_clt
            PUBLIC  _@vect06

@@BITS      BSEG                        ; Segment for booleantype and bittype variable
_b_bit      DBIT

@@R_INIS    CSEG    UNIT64KP            ; Segment for initialization data (Iinitialized
                                        ; sreg variable)
            DW      03H             ; 3

@@INIS      DSEG    SADDRP              ; Segment for tentative data area (Iinitialized
                                        ; sreg variable)
_sr_inis :  DS      ( 2 )

@@DATS      DSEG    SADDRP             ; Segment for tentative data area (Uninitialized
                                        ; sreg variable)
_sr_dats :  DS      ( 2 )
```

```
@@CNSTL     CSEG    PAGE64KP            ; Segment for const variable
_i_cnst :   DW      01H             ; 1

@@RLINIT    CSEG    UNIT64KP            ; Segment for initialization data (with initial
                                        ; value)
            DW      00002H, 00000H  ; 2

@@INITL     DSEG  UNIT64KP          ; Segment for tentative data (with initial value)
_l_init :   DS    ( 4 )

@@DATAL     DSEG  UNIT64KP              ; Segment for tentative data (without initial
                                        ; value)
_i_data :   DS    ( 2 )

@@CALT      CSEG    CALLT0              ; Segment for callt function table
?f_clt :    DW      _f_clt
; line  1 : #pragma INTERRUPT   INTP0   inter   rb1     /* Interrupt function */
; line  2 :
; line  3 : void main ( void );             /* Function prototype declaration */
; line  4 : const   int     i_cnst = 1 ;    /* const variable */
; line  5 : callt   void f_clt ( void );    /* calltFunction prototype declaration */
; line  6 : __boolean       b_bit ;         /* boolean type variable */
; line  7 : long    l_init = 2 ;            /* Initialized external variable */
; line  8 : int     i_data ;                /* Uninitialized external variable */
; line  9 : __sreg  int     sr_inis = 3 ;   /* Iinitialized sreg variable */
; line 10 : __sreg  int     sr_dats ;       /* Uninitialized sreg variable */
; line 11 :
; line 12 : void main ( ) {                 /* Function definition */

@@CODEL     CSEG                        ; Segment for code portion
_main :
        push    hl
; line 13 :int     i ;
; line 14 :i = 100 ;
        movw    hl, #064H         ; 100
; line 15 : }
        pop     hl
        ret
; line 16 :
; line 17 : void inter ( ) {                /*Interrupt function definition */

@@BASE      CSEG    BASE                ; Segment for callt and interrupt function
_inter :
; line 18 : }
        reti
; line 19 :
; line 22 : callt   void f_clt ( ) {        /* callt function definition */
_f_clt:
```

```
; line 23 : }
        ret
@@VECT06    CSEG    AT      0006H       ; Segment for vector table
_@vect06 :
            DW      _inter
            END


; Target chip : uPDxxxx
; Device file : xx.xxx
```

CHAPTER 4 ASSEMBLY LANGUAGE SPECIFICTION

This chapter explains the assembly language specifications supported by 78K0R assembler.

4.1 Description Methods of Source Program

This section explains the description methods, expressions and operators of the source program.

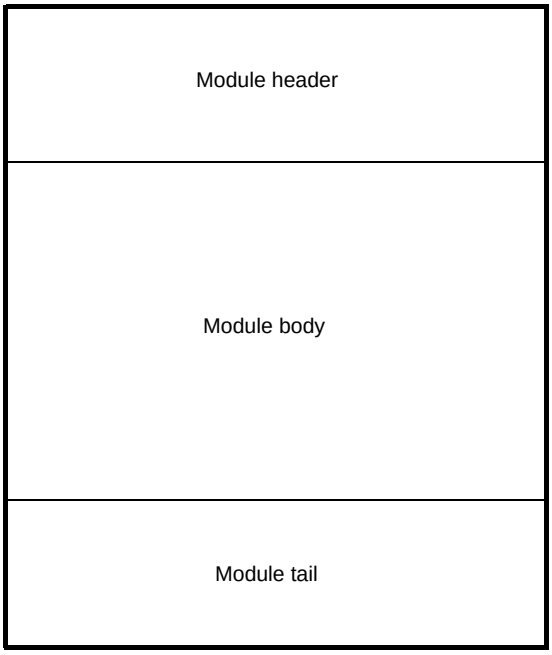
4.1.1 Basic configuration

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

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

- Module header (Module Header)
- Module body (Module Body)
- Module tail (Module Tail)

Figure 4-1. Source Module Configuration



(1) Module header

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

Also, the module header can be omitted.

Table 4-1. Instructions That Can Appear in Module Headers

Type of Instruction	Description
Control instructions with the same functions as assembler options	- PROCESSOR - DEBUG/NODEBUG/DEBUGA/NODEBUGA - XREF/NOXREF - SYMLIST/NOSYMLIST - TITLE - FORMFEED/NOFORMFEED - WIDTH - LENGTH - TAB - KANJICODE
Special control instructions output by a C compiler or other high-level program	- TOL_INF - DGS - DGL

(2) Module body

he following may not appear in the module body.

- Control instructions with the same functions as assembler options

All other directives, control instructions, and instructions can be described in the module body.
The module body must be described by dividing it into units, called " segments".
Segments are defined with the corresponding directives, as follows.

- Code segment
  Defined with the CSEG directive
- Data segment
  Defined with the DSEG directive
- Bit segment
  Defined with the BSEG directive
- Absolute segment
  Defined with the CSEG, DSEG, or BSEG directive, plus an absolute address (AT location address) as the relocation attribute. Absolute segments can also be defined with the ORG directive.

The module body may be configured as any combination of segments.
However, data segments and bit segments should be defined before code segments.

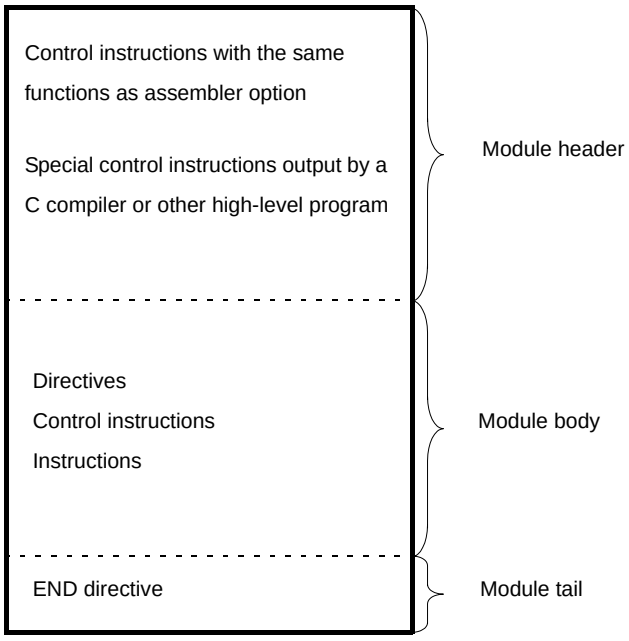
(3) Module tail

The module tail indicates the end of the source module. The END directive must be described in this part.
If anything other than a comment, a blank, a tab, or a line feed code is described following the END directive, the assembler will output a warning message and ignore the characters described after the END directive.

(4) Overall configuration of source program

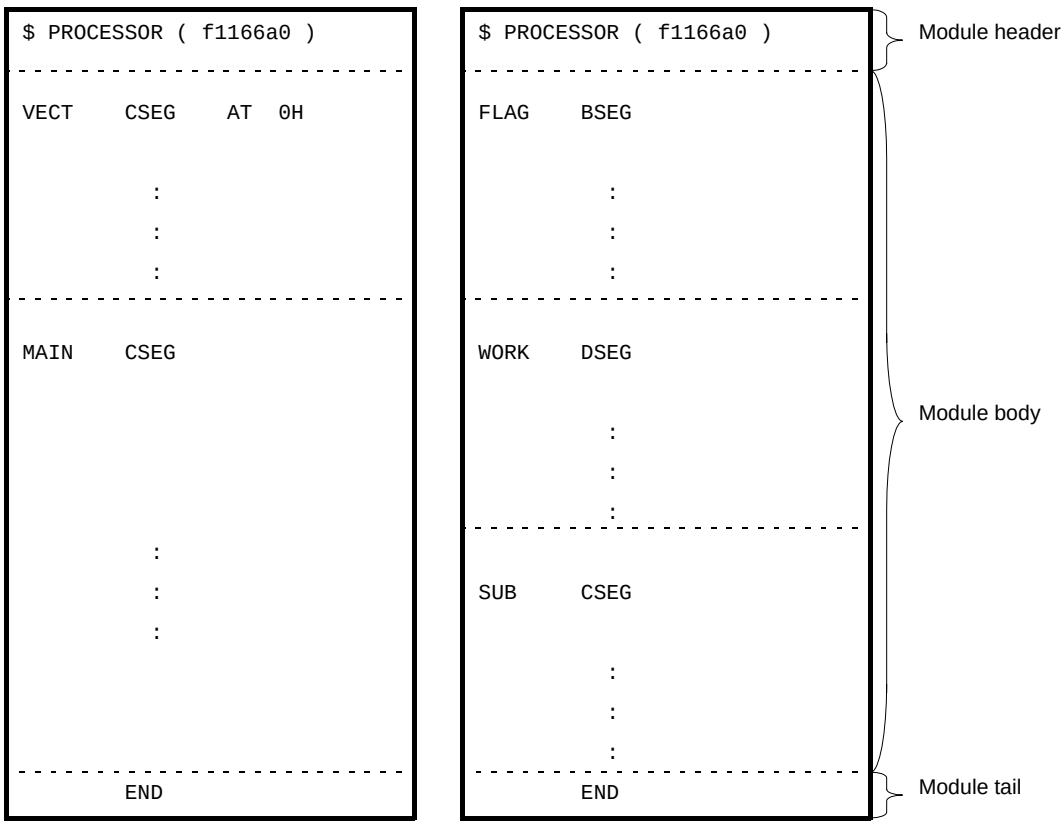
The overall configuration of a source module (source program) is as shown below.

Figure 4-2. Overall Configuration of Source Program



Examples of simple source module configurations are shown below.

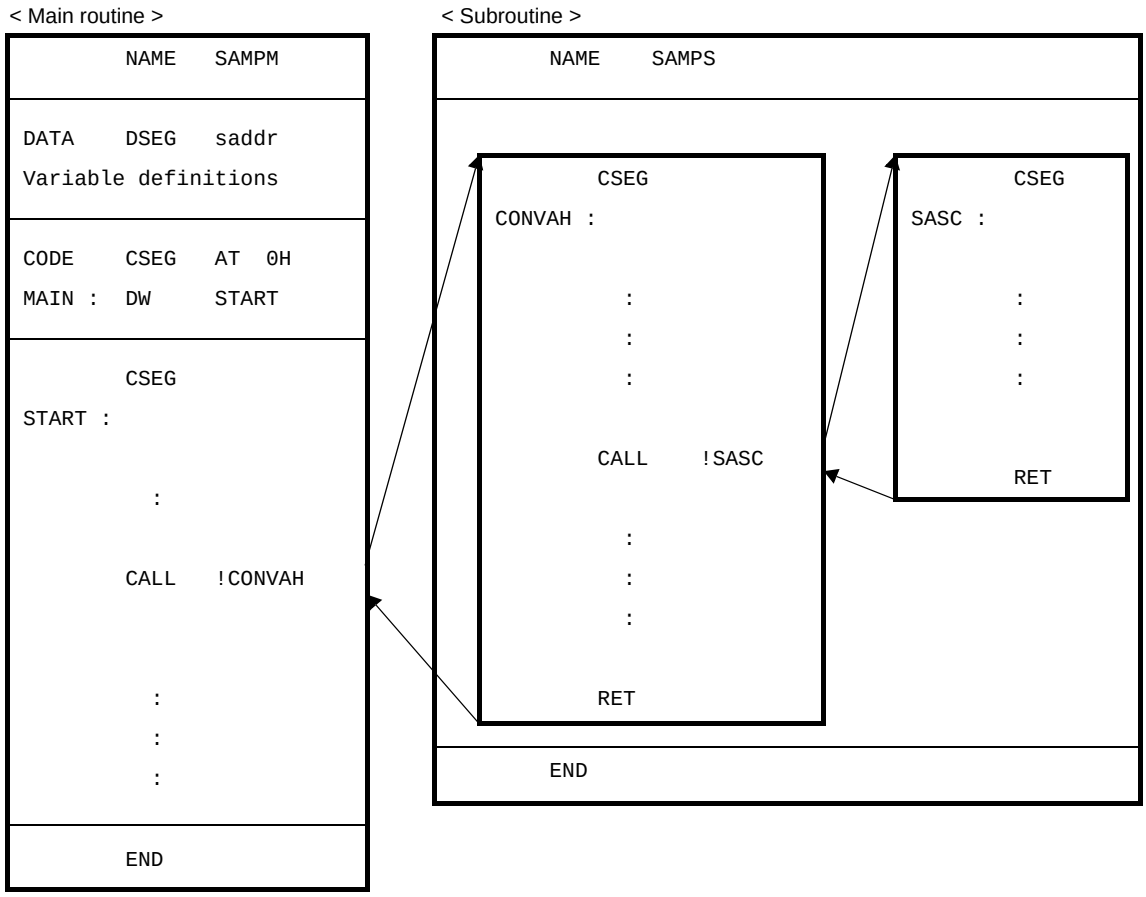
Figure 4-3. Examples of Source Module Configurations



(5) Coding example

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

Figure 4-4. Sample Source Program Configuration



- Main routine

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

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

DATA    DSEG    AT      0FFE20H         ; (5)
HDTSA : DS      1
STASC : DS      2

CODE    CSEG    AT      0H              ; (6)
```

```
MAIN :  DW      START

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

        MOV     HDTSA, #1AH
        MOVW    HL, #LOWW ( HDTSA )     ; set hex 2-code data in HL registor

        CALL    !CONVAH                 ; convert ASCII <- HEX
                                        ; output BC-register <- ASCII code
        MOVW    DE, #LOWW ( STASC )     ; set DE <- store ASCII code table
        MOV     A, B
        MOV     [DE], A
        INCW    DE
        MOV     A, C
        MOV     [DE], A
        BR      $$
        END                             ; (8)
```

(1) Declaration of module name

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

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

(4) Declaration of stack resolution symbol. This will be generated by the linker when the program is linked with the -s option specified. (An error occurs if the linker -s option is not specified.)

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

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

(7) Declaration of start of another code segment (ending the absolute code segment)

(8) Declaration of end of module

- Subroutine

```
        NAME    SAMPS           ; (9)
; ********************************************************
;       HEX -> ASCII Conversion Program
;       sub-routine
;
;   input condition     : ( HL )        <- hex 2 code
;   output condition    : BC-register   <- ASCII 2 code
; ********************************************************
```

```
PUBLIC  CONVAH                  ; (10)

        CSEG                    ; (11)
CONVAH :
        XOR     A, A
        ROL4    [HL]            ; hex upper code load (12)
        CALL    !SASC
        MOV     B, A            ; store result

        XOR     A, A
        ROL4    [HL]            ; hex lower code load
        CALL    !SASC
        MOV     C, A            ; store result
        RET

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

SASC :
        CMP     A, #0AH         ; check hex code > 9
        BC      $SASC1
        ADD     A, #07H         ; bias ( +7H )
SASC1 :
        ADD     A, #30H         ; bias ( +30H )
        RET
        END                     ; (13)
```

(9) Declaration of module name

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

(11) Declaration of start of code segment

(12) The ROL4 instruction is 78K0 instruction that is not supported by the 78K0R. The assembler -compati option must be specified to assemble this module.
For the assembler option (-compati), refer to the 78K0R build User's Manual.

(13) Declaration of end of module

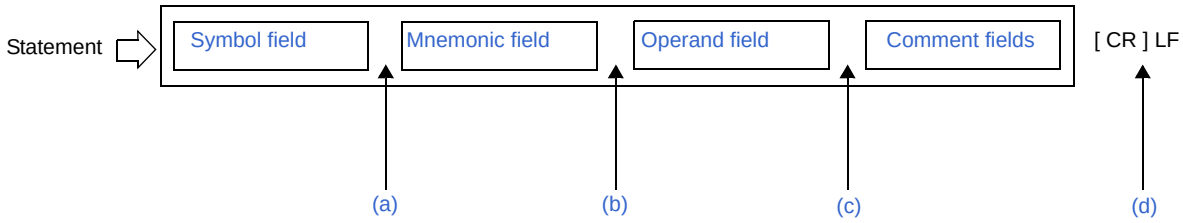
4.1.2 Description method

(1) Configuration

A source program consists of statements.

A statement is made up of the 4 fields shown below.

Figure 4-5. Statement Fields



(a) The symbol field and the mnemonic field must be separated by a colon (:) or one or more spaces or tabs. (Whether a colon or space is required depends on the instruction in the mnemonic field.)

(b) The mnemonic field and the operand field must be separated by one or more spaces or tabs. Depending on the instruction in the mnemonic field, the operand field may not be required.

(c) The comment field, if present, must be preceded by a semicolon (;).

(d) Each line in the source program ends with an LF code. (One CR code may exist before the LF code.)

- A statement must be described in 1 line. The line can be up to 2048 characters long (including CR/LF). TAB and the CR (if present) are each counted as 1 character. If the length of the line exceeds 2048 characters, a warning is issued and all characters beyond the 2048th are ignored for purposes of assembly. However, characters beyond 2048 are output to assembler list files.
- Lines consisting of CR only are not output to assembler list files.
- The following line types are valid.
  - Empty lines (lines with no statements)
  - Lines consisting of the symbol field only
  - Lines consisting of the comment field only

(2) Character set

Source files can contain the following 3 types of characters.

- Language characters
- Character data
- Comment characters

(a) Language characters

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

Table 4-2. Alphanumeric Characters

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

Table 4-3. Special Characters

Character	Name	Main Use	
?	Question mark	Symbol equivalent to alphabetic characters	
@	Circa	Symbol equivalent to alphabetic characters	
_	Underscore	Symbol equivalent to alphabetic characters	
	Blank	Field delimiter	Delimiter symbols
HT (09H)	Tab code	Character equivalent to blank	
,	Comma	Operand delimiter	
:	Colon	Label delimiter	
;	Semicolon	Symbol indicating the start of the Comment field	
CR (0DH)	Carriage return code	Symbol indicating the end of a line (ignored in the assembler)	
LF (0AH)	Line feed code	Symbol indicating the end of a line	
+	Plus sign	Add operator or positive sign	Assembler operators
-	Minus sign	Subtract operator or negative sign	
*	Asterisk	Multiply operator	
/	Slash	Divide operator	
.	Period	Bit position specifier	
( )	Left and right parentheses	Symbols specifying the order of arithmetic operations to be performed	
< >	Not equal sign	Relational operator	
=	Equal sign	Relational operator	
’	Single quote mark	- Symbol indicating the start or end of a character constant - Symbol indicating a complete macro parameter	
$	Dollar sign	- Symbol indicating the location counter - Symbol indicating the start of a control instruction equivalent to an assembler option - Symbol specifying relative addressing	
&	Ampersand	Concatenation symbol (used in macro body)	
#	Sharp sign	Symbol specifying immediate addressingr	
!	Exclamation point	Symbol specifying absolute addressing	
[ ]	Brackets	Symbol specifying indirect addressing	

(b) Character data

Character data refers to characters used to write string literals, strings, and the quote-enclosed operands of some control instructions (TITLE, SUBTITLE, INCLUDE).

Cautions 1. Character data can use all characters except 00H (including multibyte kanji, although the encoding depends on the OS). If 00H is encountered, an error occurs and all characters from the 00H to the closing single quote (') are ignored.

2. When an invalid character is encountered, the assembler replaces it with an exclamation point (!) in the assembly list. (The CR (0DH) character is not output to assembly lists.)

3. The Windows OS interprets code 1AH as an end of file (EOF) code. Input data cannot contain this code.

(c) Comment characters

Comment characters are used to write comments.

Caution Comment characters and character data have the same character set. However, no error is output for 00H in comments. 00H is replaced by "!" in assembly lists.

(3) Symbol field

The symbol field is for symbols, which are names given to addresses and data objects. Symbols make programs easier to understand.

(a) Symbol types

Symbols can be classified as shown below, depending on their purpose and how they are defined.

Symbol Type	Purpose	Definition Method
Name	Used as names for addresses and data objects in source programs.	Write in the symbol field of an EQU, SET, or DBIT directive.
Label	Used as labels for addresses and data objects in source programs.	Write a name followed by a colon (:).
External reference name	Used to reference symbols defined by other source modules.	Write in the operand field of an EXTRN or EXTBIT directive.
Segment name	Used at link time.	Write in the symbol field of a CSEG, DSEG, BSEG, or ORG directive.
Module name	Used during symbolic debugging.	Write in the operand field of a NAME directive.
Macro name	Use to name macros in source programs.	Write in the symbol field of a MACRO directive.

Caution The 4 types of symbols that can be written in symbol fields are names, labels, segment names, and macro names.

(b) Conventions of symbol description

Observe the following conventions when writing symbols.

- The characters which can be used in symbols are the alphanumeric characters and special characters (?, @, _).

  The first character in a symbol cannot be a digit (0 to 9).
- The maximum length of symbols is 256 characters.

  Characters beyond the maximum length are ignored.
- Reserved words cannot be used as symbols.

  See "4.5 Reserved Words" for a list of reserved words.
- The same symbol cannot be defined more than once.

  However, a name defined with the SET directive can be redefined with the SET directive.
- The assembler distinguishes between lowercase and uppercase characters.
- When a label is written in a symbol field, the colon (:) must appear immediately after the label name.

<Examples of correct symbols>

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

<Examples of incorrect symbols>

```
1ABC    EQU     3       ; The first character is a digit.s
LAB     MOV     A, R0   ; "LAB"is a label and must be separated from the mnemonic field by
                        ; a colon ( : ).
FLAG :  EQU     10H     ; The colon ( : ) is not needed for names.
```

<Example of a symbol that is too long>

```
A123456789B12...Y123456789Z123456       EQU     70H
            257characters           ; The last character (6) is ignored because it is
                                    ; beyond the maximum symbol length.
                                    ; The symbol is defined as
                                    ; A123456789B12...Y123456789Z12345
```

< Example of a statement composed of a symbol only>

```
ABCD :                  ; ABCD is defined as a label.
```

(c) Points to note about symbols

??Rannnn (where nnnn = 0000 to FFFF) is a symbol that the assembler replaces automatically every time it generates a local symbol in a macro body. Unless care is taken, this can result in duplicate definitions, which are errors.

The assembler generates a name automatically when a segment definition directive does not specify a name. These segment names are listed below.

Duplicate segment name definitions are errors.

Segment Name	Directive	Relocation Attribute
?A0nnnnn (nnnnn = 00000 - FFFFF)	ORG directive	(none)
?CSEG	CSEG directive	UNIT
?CSEGUP		UNITP
?CSEGT0		CALLT0
?CSEGFX		FIXED
?CSEGSI		SECUR_ID
?CSEGB		BASE
?CSEGP64		PAGE64KP
?CSEGU64		UNIT64KP
?CSEGMIP		MIRRORP
?CSEGOB0		OPT_BYTE
?DSEG	DSEG directive	UNIT
?DSEGUP		UNITP
?DSEGS		SADDR
?DSEGSP		SADDRP
?DSEGBP		BASEP
?DSEGP64		PAGE64KP
?DSEGU64		UNIT64KP
?BSEG	BSEG directive	UNIT

(d) Symbol attributes

Every name and label has both a value and an attribute.
The value is the value of the defined data object, for example a numerical value, or the value of the address itself.
Segment names, module names, and macro names do not have values.
The following table lists symbol attributes.

Attribute Type	Classification	Value
NUMBER	- Name to which numeric constants are assigned - Symbols defined with the EXTRN directive - Numeric constants	Decimal notation : 0 to 1048575 Hexadecimal notation : 00000H to FFFFFH (unsigned)
ADDRESS	- Symbols defined as labels - Names of labels defined with the EQU and SET directives	Decimal notation : 0 to 1048575 Hexadecimal notation : 0H to FFFFFH
BIT	- Names defined as bit values - Names in BSEG - Symbols defined with the EXTBIT directive	0H to FFFFFH
SFR	Names defined as SFRs with the EQU directive	SFR area
SFRP	Names defined as SFRs with the EQU directive	

Attribute Type	Classification	Value
CSEG	Segment names defined with the CSEG directive	These attribute types have no values.
DSEG	Segment names defined with the DSEG directive	
BSEG	Segment names defined with the BSEG directive	
MODULE	Module names defined with the NAME directive. (If not specified, a module name is created from the primary name of the input source filename.)	
MACRO	Macro names defined with the MACRO directive	

<Examples>

```
TEN      EQU     10H          ; The name TEN has the NUMBER attribute and a value of 10H.
         ORG     80H
START :  MOV     A, #10H      ; The label START has the ADDRESS attribute and a value of 80H.
BIT1     EQU     0FFE20H.0    ; The name BIT1 has the BIT attribute and a value of 0FFE20H.0.
```

(4) Mnemonic field

Write instruction mnemonics, directives, and macro references in the mnemonic field.

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

However, if the first operand begins with "#", "$","!", or "[", the statement will be assembled properly even if nothing exists between the mnemonic field and the first operand field.

<Examples of correct mnemonics>

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

<Examples of incorrect mnemonics>

```
MOVA    #0H             ; There is no blank between the mnemonic and operand fields.
C ALL    !CONVAH        ; The mnemonic field contains a blank.
ZZZ                     ; The 78K0R series does not have a ZZZ instruction.
```

(5) Operand field

In the operand field, write operands (data) for the instructions, directives, or macro references that require them.

Some instructions and directives require no operands, while others require two or more.

When you provide two or more operands, delimit them with a comma (,).

The following types of data can appear in the operand field:

- Constants(numeric or string)
- Character strings
- Register names
- Special characters($ # ! [ ])
- Relocation attributes of segment definition directives
- Symbols
- Expressions
- Bit terms

The size and attribute of the required operand depends on the instruction or directive. See "4.1.16 Operand characteristics" for details.
See the user's manual of the target device for the format and notational conventions of instruction set operands.
The following sections explain the types of data that can appear in the operand field.

(a) Constants

A constant is a fixed value or data item and is also referred to as immediate data.
There are numeric constants and character string constants.

- Numeric constants

Numeric constants can be written in binary, octal, decimal, or hexadecimal notation.
The table below lists the notations available for numeric constants.
Numeric constants are handled as unsigned 32-bit data.
The range of possible values is 0 <= n <= 0FFFFFFFFH.
Use the minus sign operator to indicate minus values.

Type	Notation	Example
Binary	Append a "B" or "Y" suffix to the number.	1101B 1101Y
Octal	Append an "O" or "Q" suffix to the number.	74O 74Q
Decimal	Simply write the number, or append a "D" or "T" suffix.	128 128D 128T
Hexadecimal	- Append an "H" suffix to the number. - If the number begins with "A", "B", "C", "D", "E", or "F", prefix it with "0"	8CH 0A6H

- Character string constants

A character-string constant is expressed by enclosing a string of characters from those shown in "(2) Character set", in a pair of single quotation marks (').
The assembler converts string constants to 7-bit ASCII codes, with the parity bit set to 0.
The length of a string constant is 0 to 2 characters.
To include the single quote character in the string, write it twice in succession.

<Example string constants >

```
'ab'            ; 6162H
'A'             ; 0041H
'A'''           ; 4127H
' '             ; 0020H (1 space character)
```

(b) Character strings

A character string is expressed by enclosing a string of characters from those shown in "(2) Character set", in a pair of single quotation marks (').

The main use for character strings is as operands for the DB and CALL directives and the TITLE and SUB-TITLE control instructions.

<Special character example>

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

(c) Register names

The following registers can be named in the operand field:

- General registers
- General register pairs
- Special function registers

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

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

(d) Special characters

The following table lists the special characters that can appear in the operand field.

Special Character	Function
$	- Indicates the location address of the instruction that has the operand (or the first byte of the address, in the case of multibyte instructions). - Indicates relative addressing for a branch instruction.
!	- Indicates absolute addressing for a branch instruction. - Indicates an addr16 specification, which allows a MOV instruction to access the entire memory space.
#	- Indicates immediate data.
[ ]	- Indicates indirect addressing.

<Special character example>

```
Address     Source program
100             ADD     A, #10H
102     LOOP :  INC     A
103             BR      $$ - 1      ; The second $ in the operand indicates address
                                    ; 103H.  Describing "BR $ - 1" results in the
                                    ; same operation.
105             BR      !$ + 100H   ; The second $ in the operand indicates address
                                    ; 105H.  Describing "BR $ + 100H" results in the
                                    ; same operation.
```

(e) Relocation attributes of segment definition directives

Relocation attributes can appear in the operand field.

See "4.2.2 Segment definition directives" for more information about relocation attributes.

(f) Symbols

When a symbol appears in the operand field, the address (or value) assigned to that symbol becomes the operand value.

<Symbol value example>

```
VALUE   EQU     1234H
        MOV     AX, #VALUE     ; This could also be written MOV     AX, #1234H
```

(g) Expressions

An expression is a combination of constants, location addresses (indicated by $), names, labels, and operators, which is evaluated to produce a value.

Expressions can be specified as instruction operands wherever a numeric value can be specified.

See "4.1.3 Expressions and operators" for more information about expressions.

<Expression example>

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

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

In this expression, a name and a numeric value are connected by the - (minus) operator. The value of the expression is 0BH, so this expression could be rewritten as "MOV A , #0BH".

(h) Bit terms

Bit terms are obtained by the bit position specifier.

See "4.1.14 Bit position specifier" for more information about bit terms.

<Bit term examples>

```
CLR1    A.5
SET1    1 + 0FFE30H.3   ; The value of this operand is 0FFE31H.3
CLR1    0FFE40H.4 + 2   ; The value of this operand is 0FFE40H.6
```

(6) Comment fields

Describe comments in the comment field, after a semicolon (;).

The comment field continues from the semicolon to the new line code at the end of the line, or to the EOF code of the file.

Comments make it easier to understand and maintain programs.

Comments are not processed by the assembler, and are output verbatim to assembly lists.

Characters that can be described in the comment field are those shown in "(2) Character set ".

<Comment example>

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

        PUBLIC  MAIN, START
        EXTRN   CONVAH
        EXTRN   @STBEG

DATA    DSEG    saddr
HDTSA:  DS      1
STASC:  DS      2

CODE    CSEG    AT 0H
MAIN :  DW      START

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

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

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

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

Lines with comment fields only

Lines with comment fields only

Lines with comments in comment fields

4.1.3 Expressions and operators

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

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

The assembler supports the operators shown in "Table 4-4. Operator Types". Operators have priority levels, which determine when they are applied in the calculation. The priority order is shown in "Table 4-5. Operator Precedence Levels".

The order of calculation can be changed by enclosing terms and operators in parentheses "( )".

<Example>

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

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

Table 4-4. Operator Types

Operator Type	Operators
Arithmetic operators	+, -, *, /, MOD, +sign, -sign
Logic operators	NOT, AND, OR, XOR
Relational operators	EQ (=), NE (< >), GT (>), GE (>=), LT (<), LE (<=)
Shift operators	SHR, SHL
Byte separation operators	HIGH, LOW
Word separation operators	HIGHW, LOWW
Special operators	DATAPOS, BITPOS, MASK
Other operator	( )

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

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

Table 4-5. Operator Precedence Levels

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

Expressions are operated according to the following rules.

- The order of operation is determined by the priority level of the operators.
  When two operators have the same priority level, operation proceeds from left to right, except in the case of unary operators, where it proceeds from right to left.
- Sub-expressions in parentheses "( )" are operated before sub-expressions outside parentheses.
- Operations involving consecutive unary operators are allowed.
  Examples:
  1 = - - 1 == 1
  -1 = -+1 = -1
- Expressions are operated using unsigned 32-bit values.
  If intermediate values overflow 32 bits, the overflow value is ignored.
- If the value of a constant exceeds 32 bits, an error occurs, and its value is calculated as 0.
- In division, the decimal fraction part is discarded.
  If the divisor is 0, an error occurs and the result is 0.
- Negative values are represented as two's complement.
- External reference symbols are evaluated as 0 at the time when the source is assembled (the evaluation value is determined at link time).
- The results of operating an expression in the operand field must meet the requirements of the instruction for a valid operand.
  When the expression includes a relocatable or external reference term, and the instruction requires an 8-bit operand, then object code is generated with the value of the least significant 8 bits and the information required for 16 bits is output in the relocation information. Subsequently the linker checks whether the previously determined value overflows the range of 8 bits. If it overflows, a linking error occurs.
  In the case of absolute expressions, the value is determined at the assembly stage and a check is performed at that stage to test whether the result fits in the required range.
  For example, the MOV instruction requires 8-bit operands, so the operand must be in the range 0H to 0FFH.

(1) Correct examples

```
MOV     A, #'2*' AND 0FH
MOV     A, #4 * 8 * 8 - 1
```

(2) Incorrect examples

```
MOV     A, #'2*.
MOV     A, #4 * 8 * 8
```

(3) Evaluation examples

Expression	Evaluation
2 + 4 * 5	22
(2 + 3) * 4	20
10/4	2
0 - 1	0FFFFFFFFH
-1 > 1	00H (False)
EXT[Note] + 1	1

Note EXT : External reference symbols

4.1.4 Arithmetic operators

The following arithmetic operators are available.

Operator	Overview
+	Addition of values of first and second terms
-	Subtraction of value of first and second terms
*	Multiplacation of value of first and second terms.
/	Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result.
MOD	Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term.
+sign	Returns the value of the term as it is.
-sign	The term value 2 complement is sought.

+

Addition of values of first and second terms

[Function]

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

[Application example]

```
        ORG     100H
START : BR      !$ + 6          ; (1)
```

(1) The BR instruction causes a jump to "current location address plus 6", namely, to address "100H + 6H = 106H".
Therefore, (1) in the above example can also be described as: START : BR !106H

-

Subtraction of value of first and second terms

[Function]

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

[Application example]

```
        ORG     100H
BACK :  BR      BACK - 6H       ;  (1)
```

(1) The BR instruction causes a jump to "address assigned to BACK minus 6", namely, to address "100H - 6H = 0FAH".
Therefore, (1) in the above example can also be described as: BACK : BR !0FAH

*****

Multiplacation of value of first and second terms

[Function]

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

[Application example]

```
TEN     EQU     10H
        MOV     A, #TEN * 3    ;  (1)
```

(1) With the EQU directive, the value "10H" is defined in the name "TEN".
"#" indicates immediate data. The expression "TEN * 3" is the same as "10H * 3" and returns the value "30H".
Therefore, (1) in the above expression can also be described as: MOV A , #30H

/

Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result.

[Function]

Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result.

The decimal fraction part of the result will be truncated. If the divisor (2nd term) of a division operation is 0, an error occurs

[Application example]

```
MOV     A, #256 / 50          ;  (1)
```

(1) The result of the division "256 / 50" is 5 with remainder 6.
The operator returns the value "5" that is the integer part of the result of the division.
Therefore, (1) in the above expression can also be described as: MOV A , #5

MOD

Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term.

[Function]

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

[Application example]

```
MOV     A, #256  MOD  50       ;  (1)
```

(1) The result of the division "256 / 50" is 5 with remainder 6.
The MOD operator returns the remainder 6.
Therefore, (1) in the above expression can also be described as: MOV A , #6

+sign

Returns the value of the term as it is.

[Function]

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

[Application example]

```
FIVE    EQU     +5      ;  (1)
```

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

-sign

The term value 2 complement is sought.

[Function]

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

[Application example]

```
NO      EQU     -1      ;  (1)
```

(1) -1 becomes the two's complement of 1.
The two's complement of binary 0000 0000 0000 0000 0000 0000 0000 0001 becomes:
1111 1111 1111 1111 1111 1111 1111 1111
Therefore, with the EQU directive, the value "0FFFFFFFFH" is defined in the name "NO".

4.1.5 Logic operators

The following logic operators are available.

Operator	Overview
NOT	Obtains the logical negation (NOT) by each bit.
AND	Obtains the logical AND operation for each bit of the first and second term values.
OR	Obtains the logical OR operation for each bit of the first and second term values.
XOR	Obtains the exclusive OR operation for each bit of the first and second term values.

NOT

Obtains the logical negation (NOT) by each bit.

[Function]

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

[Application example]

```
MOVW    AX, #LOWW ( NOT 3H )   ; (1)
```

(1) Logical negation is performed on "3H" as follows:

NOT)	0000	0000	0000	0000	0000	0000	0000	0011
	1111	1111	1111	1111	1111	1111	1111	1100

0FFFFFFFCH is returned.
Therefore, (1) can also be described as: MOVW AX , #LOWW #0FFFFFFFCH

AND

Obtains the logical AND operation for each bit of the first and second term values.

[Function]

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

A blank is required before and after the AND operator.

[Application example]

```
MOV     A, #6FAH  AND  0FH     ;  (1)
```

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

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

The result "0AH" is returned. Therefore, (1) in the above expression can also be described as: MOV A , #0AH

OR

Obtains the logical OR operation for each bit of the first and second term values.

[Function]

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

A blank is required before and after the OR operator.

[Application example]

```
MOV     A, #0AH  OR  1101B     ;  (1)
```

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

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

The result "0FH" is returned.

Therefore, (1) in the above expression can also be described as: MOV A , #0FH

XOR

Obtains the exclusive OR operation for each bit of the first and second term values.

[Function]

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

[Application example]

```
MOV     A, #9AH  XOR  9DH      ;  (1)
```

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

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

The result "7H" is returned.
Therefore, (1) in the above expression can also be described as: MOV A , #7H

4.1.6 Relational operators

The following relational operators are available.

Operator	Overview
EQ (=)	Compares whether value s of first term and second term are equivalent.
NE (< >)	Compares whether values of first term and second term are not equivalent.
GT (>)	Compares whether value of first term is greater than value of the second.
GE (>=)	Compares whether value of first term is greater than or equivalent to the value of the second term.
LT (<)	Compares whether value of first term is smaller than value of the second.
LE (<=)	Compares whether value of first term is smaller than or equivalent to the value of the second term.

EQ (=)

Compares whether value s of first term and second term are equivalent.

[Function]

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

A blank is required before and after the EQ operator.

[Application example]

```
A1      EQU     12C4H
A2      EQU     12C0H

        MOV     A, #A1  EQ  ( A2 + 4H )        ;  (1)
        MOV     X, #A1  EQ   A2                ;  (2)
```

(1) In (1) above, the expression "A1 EQ (A2 + 4H)" becomes "12C4H EQ (12C0H + 4H)".
The operator returns 0FFH because the value of the 1st term is equal to the value of the 2nd term.

(2) In (2) above, the expression "A1 EQ A2" becomes "12C4H EQ 12C0H".
The operator returns 00H because the value of the 1st term is not equal to the value of the 2nd term.

NE (< >)

Compares whether values of first term and second term are not equivalent.

[Function]

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

A blank is required before and after the NE operator.

[Application example]

```
A1      EQU     5678H
A2      EQU     5670H

        MOV     A, #A1  NE   A2              ;  (1)
        MOV     A, #A1  NE  ( A2 + 8H )      ;  (2)
```

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

(2) In (2) above, the expression "A1 NE (A2 + 8H)" becomes "5678H NE (5670H + 8H)".
The operator returns 00H because the value of the 1st term is equal to the value of the 2nd term.

GT (>)

Compares whether value of first term is greater than value of the second.

[Function]

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

A blank is required before and after the GT operator.

[Application example]

```
A1      EQU     1023H
A2      EQU     1013H

        MOV     A, #A1  GT   A2              ;  (1)
        MOV     X, #A1  GT  ( A2 + 10H )     ;  (2)
```

(1) In (1) above, the expression "A1 GT A2" becomes "1023H GT 1013H".
The operator returns 0FFH because the value of the 1st term is greater than the value of the 2nd term.

(2) In (2) above, the expression "A1 GT (A2 + 10H)" becomes "1023H GT (1013H + 10H)".
The operator returns 00H because the value of the 1st term is equal to the value of the 2nd term.

GE (>=)

Compares whether value of first term is greater than or equivalent to the value of the second term.

[Function]

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

A blank is required before and after the GE operator.

[Application example]

```
A1      EQU     2037H
A2      EQU     2015H

        MOV     A, #A1  GE   A2              ;  (1)
        MOV     X, #A1  GE  ( A2 + 23H )     ;  (2)
```

(1) In (1) above, the expression "A1 GE A2" becomes "2037H GE 2015H".
The operator returns 0FFH because the value of the 1st term is greater than the value of the 2nd term.

(2) In (2) above, the expression "A1 GE (A2 + 23H)" becomes "2037H GE (2015H + 23H)".
The operator returns 00H because the value of the 1st term is less than the value of the 2nd term.

LT (<)

Compares whether value of first term is smaller than value of the second.

[Function]

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

A blank is required before and after the LT operator

[Application example]

```
A1      EQU     1000H
A2      EQU     1020H

        MOV     A, #A1  LT  A2              ;  (1)
        MOV     X, # ( A1 + 20H )  LT  A2   ;  (2)
```

(1) In (1) above, the expression "A1 LT A2" becomes "1000H LT 1020H".
The operator returns 0FFH because the value of the 1st term is less than the value of the 2nd term.

(2) In (2) above, the expression "(A1 + 20H) LT A2" becomes "(1000H + 20H) LT 1020H".
The operator returns 00H because the value of the 1st term is equal to the value of the 2nd term.

LE (<=)

Compares whether value of first term is smaller than or equivalent to the value of the second term.

[Function]

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

A blank is required before and after the LE operator.

[Application example]

```
A1      EQU     103AH
A2      EQU     1040H

        MOV     A, #A1  LE  A2              ;  (1)
        MOV     X, # ( A1 + 7H )  LE  A2    ;  (2)
```

(1) In (1) above, the expression "A1 LE A2" becomes "103AH LE 1040H".
The operator returns 0FFH because the value of the 1st term is less than the value of the 2nd term.

(2) In (2) above, the expression "(A1 + 7H) LE A2" becomes "(103AH + 7H) LE 1040H".
The operator returns 00H because the value of the 1st term is greater than the value of the 2nd term.

4.1.7Shift operators

The following shift operators are available.

Operator	Overview
SHR	Obtains only the right-shifted value of the first term which appears in the second term.
SHL	Obtains only the left-shifted value of the first term which appears in the second term.

SHR

Obtains only the right-shifted value of the first term which appears in the second term.

[Function]

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

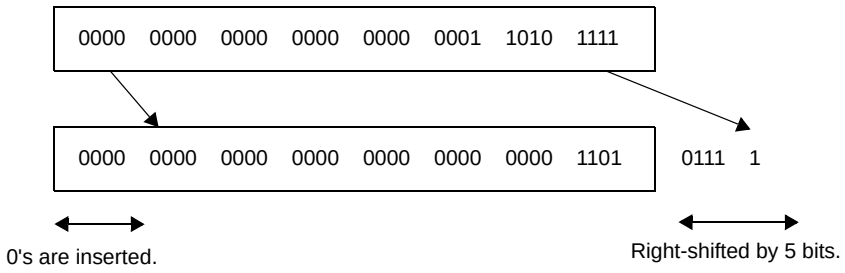
A blank is required before and after the SHR operator.

If the number of shifted bits is 0, the value of the first term is returned as is. If the number of shifted bits exceeds 32, the space is automatically filled with zeros.

[Application example]

```
MOV     A, #01AFH  SHR  5      ;  (1)
```

(1) This operator shifts the value "01AFH" to the right by 5 bits.



The value "000DH" is returned.

Therefore, (1) in the above example can also be described as: MOV A , #0DH

SHL

Obtains only the left-shifted value of the first term which appears in the second term.

[Function]

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

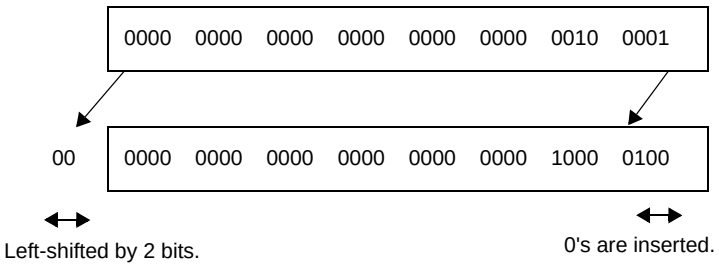
A blank is required before and after the SHL operator.

If the number of shifted bits is 0, the value of the first term is returned as is. If the number of shifted bits exceeds 32, the space is automatically filled with zeros.

[Application example]

```
MOV     A, #21H  SHL  2        ;  (1)
```

(1) This operator shifts the value "21H" to the left by 2 bits.



The value "84H" is returned.

Therefore, (1) in the above example can also be described as: MOV A , #84H

4.1.8 Byte separation operators

The following byte separation operators are available.

Operator	Overview
HIGH	Returns the high-order 8-bit value of a term.
LOW	Returns the low-order 8-bit value of a term.

HIGH

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

[Function]

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

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

[Application example]

```
MOV     A, #HIGH  1234H        ;  (1)
```

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

Therefore, (1) in the above example can also be described as: MOV A , #12H

[Remark]

A HIGH operation for an SFR name is performed, using either of the following description methods.

```
HIGHΔSFR-name
```

Or,

```
HIGH[Δ]([Δ]SFR-name[Δ])
```

The result obtained from the operation is an operand of the absolute NUMBER attribute.

No other operations can be performed for the SFR name.

<Example>

```
Symbol field    Mnemonic field  Operand field
                MOV             R0, #HIGH PM0
                MOV             R1, #HIGH PM1 + 1H      ; Equivalent to ( HIGH PM1 ) + 1
                MOV             R1, #HIGH ( PM1 + 1H )  ; An error is returned because
                                                        ; operands other than HIGH, LOW,
                                                        ; HIGHW, and  LOWW are specified
                                                        ; as the SFR name
```

LOW

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

[Function]

Returns the low-order 8-bit value of a term.
A blank is required between the LOW operator and the term.

[Application example]

```
MOV     A, #LOW  1234H        ;  (1)
```

(1) By executing a MOV instruction, this operator returns the low-order 8-bit value "34H" of the expression "1234H".
Therefore, (1) in the above example can also be described as: MOV A , #34H

[Remark]

A LOW operation for an SFR name is performed, using either of the following description methods.

```
LOWΔSFR-name
```

Or,

```
LOW[Δ]([Δ]SFR-name[Δ])
```

The result obtained from the operation is an operand of the absolute NUMBER attribute.
No other operations can be performed for the SFR name.
<Example>

```
Symbol field    Mnemonic field  Operand field
                MOV             R0, #LOW PM0
                MOV             R1, #LOW PM1 + 1H       ; Equivalent to #( LOW PM1 ) + 1
                MOV             R1, #LOW ( PM1 + 1H )   ; An error is returned because
                                                        ; operands other than HIGH, LOW,
                                                        ; HIGHW, and LOWW are specified
                                                        ; as the SFR name
```

4.1.9 Word separation operators

The following word separation operators are available.

Operator	Overview
HIGHW	Returns the high-order 16-bit value of a term.
LOWW	Returns the low-order 16-bit value of a term.

HIGHW

Returns the high-order 16-bit value of a term.

[Function]

Returns the high-order 16-bit value of a term.
A blank is required between the HIGHW operator and the term.

[Application example]

```
MOVW    AX, #HIGHW     12345678H       ; (1)

MOV     ES, #HIGHW     LAB             ; (2)
MOVW    AX, ES:!LAB
```

(1) By executing a MOVW instruction, this operator returns the high-order 16-bit value "1234H" of the expression "12345678H".
Therefore, (1) in the above example can also be described as: MOVW AX , #1234H

(2) By executing the MOV instruction on line (2), the higher address of label LAB is set to the ES register.

[Remark]

A HIGHW operation for an SFR name is performed, using either of the following description methods.

```
HIGHW∆SFR-name
```

Or,

```
HIGHW[∆]([∆]SFR-name[∆])
```

The result obtained from the operation is an operand of the absolute NUMBER attribute.
No other operations can be performed for the SFR name.
<Example>

```
Symbol field    Mnemonic field  Operand field
                MOVW            RP0, #HIGHW PM0
                MOVW            RP1, #HIGHW PM1 + 1H    ; Equivalent to #( HIGHW PM1 ) + 1
                MOVW            RP1, #HIGHW ( PM1 + 1H ); An error is  returned because
                                                        ; operands other than HIGH, LOW,
                                                        ; HIGHW, and LOWW are specified
                                                        ; as the SFR name
```

LOWW

Returns the low-order 16-bit value of a term.

[Function]

Returns the low-order 16-bit value of a term.

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

[Application example]

```
MOVW    AX, #LOWW      12345678H       ; (1)
```

(1) By executing a MOV instruction, this operator returns the low-order 16-bit value "5678H" of the expression "12345678H".
Therefore, (1) in the above example can also be described as: MOVW AX , #5678H

[Remark]

A LOWW operation for an SFR name is performed, using either of the following description methods.

```
LOWWΔSFR-name
```

Or,

```
LOWW[Δ]([Δ]SFR-name[Δ])
```

The result obtained from the operation is an operand of the absolute NUMBER attribute.

No other operations can be performed for the SFR name.

<Example>

```
Symbol field    Mnemonic field  Operand field
                MOVW            RP0, #LOWW PM0
                MOVW            RP1, #LOWW PM1 + 1H     ; Equivalent to #( LOWW PM1 ) + 1
                MOVW            RP1, #LOWW ( PM1 + 1H ) ; An error is returned because
                                                        ; operands other than HIGH, LOW,
                                                        ; HIGHW, and LOWW are specified
                                                        ; as the SFR name
```

4.1.10 Special operators

The following special operators are available.

Operator	Overview
DATAPOS	Obtains the address part of a bit symbol.
BITPOS	Obtains the bit part of a bit symbol.
MASK	Obtains a 16-bit value in which the specified bit position s are 1 and all others are 0.

DATAPOS

Obtains the address part of a bit symbol.

[Function]

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

[Application example]

```
SYM     EQU     0FE68H.6              ;  (1)

        MOV     A, !DATAPOS  SYM      ;  (2)
```

(1) An EQU directive defines the name "SYM" with a value of 0FE68H.6.

(2) "DATAPOS SYM" represents "DATAPOS 0FE68H.6", and "0FE68H" is returned.
Therefore, (2) in the above example can also be described as: MOV A , !0FE68H

BITPOS

Obtains the bit part of a bit symbol.

[Function]

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

[Application example]

```
SYM     EQU     0FE68H.6              ;  (1)

        CLR1    [HL].BITPOS  SYM      ;  (2)
```

(1) An EQU directive defines the name "SYM" with a value of 0FE68H.6.

(2) "BITPOS.SYM" represents "BITPOS 0FE68H.6", and "6" is returned.
A CLR1 instruction clears [HL].6 to 0.

MASK

Obtains a 16-bit value in which the specified bit position s are 1 and all others are 0.

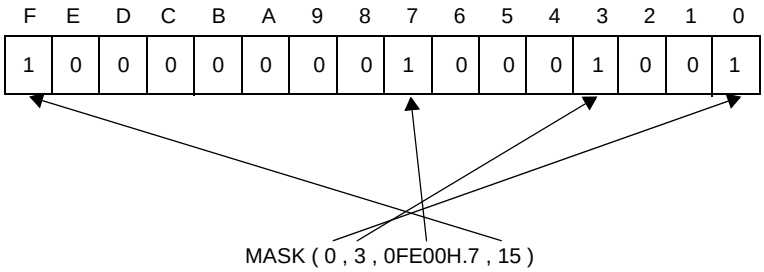
[Function]

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

[Application example]

```
MOVW    AX, #MASK ( 0, 3, 0FE00H.7, 15 )    ;  (1)
```

(1) A **MOVW instruction returns the value "8089H".**



4.1.11 Other operator

The following operators are also available.

Operator	Overview
( )	Prioritizes the calculation within ( )

( )

Prioritizes the calculation within ( )

[Function]

Causes an operation in parentheses to be performed prior to operations outside the parentheses.
This operator is used to change the order of precedence of other operators.
If parentheses are nested at multiple levels, the expression in the innermost parentheses will be calculated first.

[Application example]

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

Calculations are performed in the order of expressions (1), (2) and the value "14" is returned as a result.
If parentheses are not used,

Calculations are performed in the order (1), (2) shown above, and the value "10" is returned as a result.
See Table 4-5. Operator Precedence Levels, for the order of precedence of operators.

4.1.12 Restrictions on operations

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

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

(1) Operators and relocation attributes

Each term constituting an expression has a relocation attribute and a symbol attribute.

If terms are categorized by relocation attribute, they can be divided into 3 types: absolute terms, relocatable terms and external reference terms.

The following table shows the types of relocation attributes and their properties, and also the corresponding terms.

Table 4-6. Relocation Attribute Types

Type	Property	Corresponding Elements
Absolute term	Term that is a value or constant determined at assembly time	- Constants - Labels in absolute segments - $, indicating a location address defined in an absolute segment - Names defined with absolute values such as constants or the labels and $ listed above.
Relocatable term	Term with a value that is not determined at assembly time	- Labels defined in relocatable segments - $, indicating a relocatable address defined in a relocatable segment - Names defined with relocatable symbols
External reference term[Note]	Term for external reference of symbol in other module	- Labels defined with EXTRN directive - Names defined with EXTBIT directive

Note There are 6 operators which can take an external reference term as the target of an operation; these are "+", "-", "HIGH", "LOW", "HIGHW" and "LOWW". However, only one external reference term is allowed in one expression, and it must be connected with the "+" operator.

The following tables categorize combinations of operators and terms which can be used in expressions by relocation attribute.

Table 4-7. Combinations of Operators and Terms by Relocation Attribute (Relocatable Terms)

Operator Type	Relocation Attribute of Term			
	X:ABS Y:ABS	X:ABS Y:REL	X:REL Y:ABS	X:REL Y:REL
X + Y	A	R	R	-
X - Y	A	-	R	A[Note 1]
X * Y	A	-	-	-
X / Y	A	-	-	-
X MOD Y	A	-	-	-
X SHL Y	A	-	-	-
X SHR Y	A	-	-	-

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

ABS : Absolute term
REL : Relocatable term
A : Result is absolute term
R : Result is relocatable term
- : Operation not possible

Notes 1. Operation is possible only when X and Y are defined in the same segment, and when X and Y are not relocatable terms operated on by HIGH, LOW, HIGHW, LOWW, or DATAPOS.

2. Operation is possible when X and Y are not relocatable terms operated on by HIGH, LOW, HIGHW, LOWW, or DATAPOS.

There are 6 operators which can take an external reference term as the target of an operation; these are "+", "-", "HIGH", "LOW", "HIGHW" and "LOWW". However, only one external reference term is allowed in one expression, and it must be connected with the "+" operator.
The possible combinations of operators and terms are as follows, categorized by relocation attribute.

Table 4-8. Combinations of Operators and Terms by Relocation Attribute (External Reference Terms)

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

ABS : Absolute term
EXT : External reference term
REL : Relocatable term
A : Result is absolute term
E : Result is external reference term
R : Result is relocatable term
- : Operation not possible

Notes 1. Operation is possible only when X and Y are not external reference terms operated on by HIGH, LOW, HIGHW, LOWW, DATAPOS, or BITPOS.

2. Operation is possible only when X and Y are not relocatable terms operated on by HIGH, LOW, HIGHW, LOWW, or DATAPOS.

(2) Operators and symbol attributes

Each of the terms constituting an expression has a symbol attribute in addition to a relocation attribute.

If terms are categorized by symbol attribute, they can be divided into two types: NUMBER terms and ADDRESS terms.

The following table shows the types of symbol attributes used in expressions and the corresponding terms.

Table 4-9. Symbol Attribute Types in Operations

Symbol Attribute Type	Corresponding Terms
NUMBER term	- Symbol with NUMBER attribute - Constant
ADDRESS term	- Symbol with ADDRESS attribute - "$", indicating the location counter

The possible combinations of operators and terms are as follows, categorized by symbol attribute.

Table 4-10. Combinations of Operators and Terms by Symbol Attribute

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

ADDRES : ADDRESS term
NUMBER : NUMBER term
A : Result is ADDRESS term
N : Result is NUMBER term
- : Operation not possible

(3) How to check operation restrictions

The following is an example of how to interpret the operation of relocation attributes and symbol attributes.

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

Here, "TABLE" is presumed to be a defined label in a relocatable code segment.

(a) Operation and relocation attributes

"TABLE + 5H" is a relocatable term + an absolute term, so the rules of "Table 4-7. Combinations of Operators and Terms by Relocation Attribute (Relocatable Terms)" apply.

Operator type ... X + Y
Relocation attribute of term ... X:REL, Y:ABS

Therefore, it can be understood that the result is "R", or more specifically a relocatable term.

(b) [Operation and symbol attributes]

"TABLE + 5H" is an ADDRESS term + a NUMBER term, so the rules of "Table 4-10. Combinations of Operators and Terms by Symbol Attribute" apply.

Operator type ... X + Y
Relocation attribute of term ... X:ADDRESS, Y:NUMBER

Therefore, it can be understood that the result is "A", or more specifically an ADDRESS term.

4.1.13 Absolute expression definitions

Absolute expressions are expressions with values determined by evaluation at assembly time.
The following belong to the category of absolute expressions:

- Constants
- Expressions that are composed only of constants (constant expressions)
- Constants, EQU symbols defined from constant expressions, and SET symbols
- Expressions that operate on the above

Remark Only backward referencing of symbols is possible.

4.1.14 Bit position specifier

Bit access becomes possible via use of the (.) bit position specifier.

(1) Description Format

```
X[Δ].[Δ]Y
└──────┘
Bit term
```

X (First term)		Y (Second term)
General register	A	Expression (0 - 7)
Control register	PSW	Expression (0 - 7)
Special function register	sfr[Note]	Expression (0 - 7)
Memory	[HL][Note]	Expression (0 - 7)

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

(2) Function

- The first term specifies a byte address, and the second term specifies a bit position. This makes it possible to access a specific bit.

(3) Explanation

- An expression that uses a bit position specifier is called a bit term.
- The bit position specifier is not affected by the precedence order of operators. The left side is recognized as term 1 and the right side is recognized as term 2.

- The following restrictions apply to the first term:
  - A NUMBER or ADDRESS attribute expression, an SFR name supporting 8 -bit access, or a register name (A) can be specified.
  - If the first term is an absolute expression, the area must be 0H to 0FFFFFH.
  - External reference symbols can be specified.
- The following restrictions apply to the second term:
  - The value of the expression must be in the range from 0 to 7. When this range is exceeded, an error occurs.
  - It is possible to specify only absolute NUMBER attribute expressions.
  - External reference symbols cannot be specified.

(4) Operations and relocation attributes

- The following table shows combinations of terms 1 and 2 by relocation attribute.

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

ABS : Absolute term
REL : Relocatable term
EXT : External reference term
A : Result is absolute term
E : Result is external reference term
R : Result is relocatable term
- : Operation not possible

(5) Bit symbol values

- When a bit symbol is defined by using the bit position specifier in the operand field of an EQU directive, the value of the bit symbol is as follows:

Operand Type	Symbol Value
A.bit[Note 2]	1.bit
PSW.bit[Note 2]	FFFFAH.bit
sfr[Note 1].bit[Note 2]	FFFXXH.bit[Note 3]
expression.bit[Note 2]	XXXXXH.bit[Note 4]

Notes 1. For a detailed description, see the user's manual of each device.
2. bit = 0 - 7
3. FFFXXH is an sfr address
4. XXXXXH is an expression value

(6) Example

```
SET1    0FFE20H.3
SET1    A.5
CLR1    P1.2
SET1    1 + 0FFE30H.3    ; Equals 0FFE31H.3
SET1    0FFE40H.4 + 2    ; Equals 0FFE40H.6
```

4.1.15 Identifiers

An identifier is a name used for symbols, labels, macros etc.

Identifiers are described according to the following basic rules.

- Identifiers consist of alphanumeric characters and symbols that are used as characters (?,@,_)
  However, the first character cannot be a number (0 to 9).
- Reserved words cannot be used as identifiers.
  With regard to reserved words, see "4.5 Reserved Words".
- The assembler distinguishes between uppercase and lowercase.

4.1.16 Operand characteristics

Instructions and directives requiring one or more operands differ in the size and address range of the required operand values and in the symbol attributes of the operands.

For example, the function of the instruction "MOV r , #byte" is to transfer the value indicated by "byte" to register "r". Because the register is an 8-bit register, the data size of "byte" must be 8 bits or less.

An assembly error will occur at the statement "MOV R0 , #100H", because the value of the second operand (100H) cannot be expressed with 8 bits.

Therefore, it is necessary to bear the following points in mind when describing operands.

- Whether the size and address range are suitable for an operand of that instruction (numeric value, name, label)
- Whether the symbol attribute is suitable for suitable for an operand of that instruction (name, label)

(1) Operand value sizes and address ranges

There are conditions that limit the size and address ranges of numeric values, names and labels used as instruction operands.

For instructions, the size and address range of operands are limited by the operand representation. For directives, they are limited by the directive type.

These limiting conditions are as follows.

Table 4-11. Instruction Operand Value Ranges

Operand Representation	Value Range	
byte	8-bit value : 0H to 0FFH	
word	word [B] word [C] word [BC]	(1) Numeric constants and NUMBER attribute symbols 0H to FFFFH (2) ADDRESS attribute symbols In either of the following areas - F0000H to FFFFFH - When MAA=0, the area (01000H to 0xxxxH) mirrored to RAM space, and when MAA=1 the area (11000H to 1xxxxH) Note 1 mirrored to RAM space
	ES : word [B] ES : word [C] ES : word [BC]	(1) Numeric constants and NUMBER attribute symbols 0H to FFFFH (2) ADDRESS attribute symbols 0H to FFFFFH
	Other than the above	16-bit value : 0H to FFFFH
saddr	FFE20H to FFF1FH Note 4	
saddrp	FFE20H to FFF1FH even numbe Note 4	
sfr	FFF20H to FFFFFH : Special function register symbols (SFR symbols), numeric constants, and NUMBER attribute symbols Note 5	
sfrp	FFF20H to FFFFFH : Special function register symbols (symbols of SFRs that support 16-bit operations, even values only), numeric constants, and NUMBER attribute symbols Note 5	

Operand Representation	Value Range	
addr20	!!addr20	0H to FFFFFH
	$addr20	0H to FFFFFH, and when a branch destination is in the range (-80H) to (+7FH) from the next address after a branch or call instruction
	$!addr20	0H to FFFFFH , and when a branch destination is in the range (-8000H) to (+7FFFH) from the next address after a branch or call instruction
addr16	!addr16 (BR, CALL instructions)	0H to FFFFH (The range in which numeric constants and symbols can be specified is the same)
	!addr16[Note 2] (Other than BR, CALL instructions)	(1) Numeric constants and NUMBER attribute symbol[Note 3] 0H to FFFFH (2) ADDRESS attribute symbol[Note 3] Within one of the following: - F0000H to FFFFFH - The area mirrored to RAM space when MAA=0 (e.g. 01000H to 0xxxxH), or the area mirrored to RAM space when MAA=1 (e.g. 11000H to 1xxxxH[Note 1]
	ES:!addr16	(1) Numeric constants or NUMBER attribute symbols[Note 3] 0H to FFFFH (2) ADDRESS attribute symbols[Note 3] 0H to FFFFFH
	!addr16.bit	(1) DBIT symbol, SFBIT or SABIT attribute bit symbols, bit symbols defined with EQU directives (but only when operand includes an ADDRESS attribute symbol) Within one of the following: - F0000H to FFFFFH - The area mirrored to RAM space when MAA=0 (e.g. 01000H to 0xxxxH), or the area mirrored to RAM space when MAA=1 (e.g. 11000H to 1xxxxH)[Note 1] (2) Bit symbols other than the above 0H to FFFFH
	ES : !addr16.bit	(1) DBIT symbols, SFBIT or SABIT attribute bit symbols, bit symbols defined with EQU directives (only when operand includes an ADDRESS attribute symbol) 0H to FFFFFH (2) Bit symbols other than the above 0H to FFFFH
addr5	0080H to 00BFH (CALLT table area, even values only)	
bit	3-bit value : 0 to 7	
n	2-bit value : 0 to 3	

Notes 1. The address range of the area mirrored to RAM space differs depending to the device. For details, refer to the user's manual of the target device.

2. To describe sfr or 2ndsfr as an operand, it can be specified as !sfr and !2ndsfr. These are output as the operands for !addr16 in the code.
It is possible to described 2ndsfr without "!". The same !addr16 operand code will be output.

3. Only even addresses can be specified for 16-bit data.

4. In order to maintain compatibility with the 78K0, the range from FE20H to FF1FH can be specified with numeric constants and NUMBER attribute symbols only.

5. For numeric constants and NUMBER attribute symbols, no check of read/write access for the SFR at the specified address is performed.

The following examples explain why the symbol attribute of an addr16 or word operand affects the range that can be specified for that operand.
For more information about symbol attributes, see "(d) Symbol attributes".

(a) !addr16 (Instructions other than BR,CALL)

This section explains why the range of values that can be specified for an !addr16 operand (instructions other than BR and CALL) differs between (1) numeric constants and NUMBER attribute symbols and (2) ADDRESS attribute symbols.
Following is an example.

```
NUMBER0         EQU     0F100H                  ; (a)
NUMBER1         EQU     0F102H
NUMBER2         EQU     0F103H

D0              DSEG    AT  0FF100H
ADDRESS0:       DS      1
ADDRESS1:       DS      1
ADDRESS2:       DS      1

                CSEG
                MOV     !NUMBER0, A             ; (b)
                MOV     !0F100H, A              ; (c)
                MOV     !ADDRESS0, A            ; (d)
```

Line (a) contains a NUMBER attribute symbol. The following explains the case when this NUMBER attribute symbol is specified as an !addr16 operand.
The "MOV !addr16, A" instruction in the instruction set uses direct addressing for the !addr16 operand. In line (b) of the example, the value in register A is transferred to address 0FF100H. The NUMBER attribute symbol in line (a) could be replaced with the value in line (c). That is, the NUMBER0 symbol (the NUMBER attribute symbol specified for the !addr16 operand) and the numeric value 0F100H both indicate the same address, namely "0F100H". With respect to the range, NUMBER attribute symbols used as !addr16 operands (instructions other than BR and CALL) can have values from 0H to FFFFH. These values specify addresses from F0000H to FFFFFH.

Next, the following explains the case where the same kind of processing is performed for the ADDRESS0 label, an ADDRESS attribute symbol.
The addr16 range is 0000H to FFFFH, while the value of the ADDRESS symbol in line (d) is in the RAM memory space FxxxxH to FFFFFH. Normally this would result in an error. Therefore, to facilitate program development, provision is made for operand labels like ADDRESS0 (ADDRESS attribute symbols), under which the operand range F0000H to FFFFFH is allowed.
To summarize, !addr16 operands (instructions other that BR,CALL) that are ADDRESS attribute symbols can have values from F0000H to FFFFF. This allows them to be specified as !addr16 operands just as they are.

Additionally, support for !addr16 is required when ROM areas are mirrored to the RAM area.
This is shown in the following example.

```
M0              CSEG    MIRRORP
ADDRESS0:       DB      12H
ADDRESS1:       DB      34H
ADDRESS2:       DB      56H
                CSEG
                MOV     A, !ADDRESS0            ; (e)
```

Segment MO is located in ROM space that is mirrored to RAM space. Segment MO is located at 01000H to 0xxxxH when MAA=0, and at 11000H to 0xxxxH when MAA=1. Due to this, the value of the ADDRESS0 symbol in line (e) is in the range from 01000H to 0xxxxH or from 11000H to 1xxxxH. To facilitate program development, references to symbols in mirrored segments like the symbol in line (e) are allowed. Their !addr16 range is 01000H to 0xxxxH, or 11000H to 1xxxxH.
To summarize, !addr16 (instructions other than BR,CALL) symbols with the ADDRESS attribute can have values from 01000H to 0xxxxH, or from 11000H to 0xxxxH. This allows them to be specified as !addr16 operands just as they are.

(b) ES:!addr16

This section explains why the range of values that can be specified for an ES:!addr16 operand varies between (1) numeric constants and NUMBER attribute symbols and (2) ADDRESS attribute symbols. Following is an example.

```
DATA            CSEG    AT      12345H
ADDRESS0:       DB      12H
ADDRESS1:       DB      34H
ADDRESS2:       DB      56H


                CSEG
                MOV     ES, #HIGHW ADDRESS0     ; (f)
                MOV     A, ES:!ADDRESS0         ; (g)
```

The statements in lines (f) and (g) transfer data from ADDRESS0 to register A.
The addr16 range is 0000H to FFFFH. But in line (g) the value of the ADDRESS0 symbol is 12345H. Normally this would result in an error.
Therefore, to facilitate program development, provision is made to allow ADDRESS0 to be in the range 0H to FFFFFH, making it possible to write lines like line (g).
To summarize, ES:!addr16 operands which are ADDRESS attribute symbols can be specified just as they are. Values from 0H to FFFFFH can also be specified just as they are.

(c) !addr16.bit, ES:!addr16.bit

This section explains why the value range in !addr16.bit and ES:!addr16.bit operands differs between (1) DBIT symbols, SFBIT and SABIT attribute bit symbols, bit symbols defined by EQU directives (but only when an ADDRESS attribute symbol is included in the operand) and (2) all other symbols.
This is shown by the following example.

```
                BSEG
DBITSYM0        DBIT                            ; (h)
DBITSYM1        DBIT
DBITSYM2        DBIT

BIT1_PM0        EQU     PM0.1                   ; (i)
BIT2_P0         EQU     P0.2                    ; (j)

                DSEG
ADDRESS0:       DS      1
ADDRESS1:       DS      1
ADDRESS2:       DS      1

ADR_BIT0        EQU     ADDRESS0.0              ; (k)
ADR_BIT1        EQU     ADDRESS0.1
ADR_BIT2        EQU     ADDRESS0.2

                CSEG
                SET1    !DBITSYM0               ; (l)
                SET1    !BIT1_PM0               ; (m)
                SET1    !BIT2_P0                ; (n)
                SET1    !ADR_BIT0               ; (o)
```

Describing of the DBIT symbol on line (h), SFBIT attribute and SABIT attribute bit symbols on lines (i) and (j), and the bit symbol defined with the EQU directive on line (k) (only when an ADDRESS attribute symbol is included as an operand) as operands for !addr16.bit is made possible, as stated on lines (l) to (o), so the range of values varies depending on the symbol attribute described.
For the same reasons, the value range for ES:!addr16.bit operands also depends on the symbol attribute.

(d) word

This section explains why the value ranges of word operands differs between (1) numeric constants and NUMBER attribute symbols and (2) ADDRESS attribute symbols.

This is shown by the following example.

```
                DSEG
ADDRESS0:       DS      1
ADDRESS1:       DS      1
ADDRESS2:       DS      1
                CSEG
                MOV     B, #0
                MOV     ADDRESS0[B], A          ; (p)
                MOV     C, #1
                MOV     ADDRESS0[C], A          ; (q)
                MOVW    BC, #2
                MOV     ADDRESS0[BC], AX        ; (r)
```

Since labels (ADDRESS attribute symbols) are often specified where a word is required in an operand, such as in the word[B], word[C] and word[BC] instructions in lines (p) to (r), coding is made simpler by the ability to specify labels, in the same manner as !addr16.

In the same reason, coding of ES:word[B], ES:word[C], ES:word[BC] instructions is simplified.

Table 4-12. Value ranges of Directive Operands

Directive Type	Directive	Value Range
Segment definition	CSEG AT	0H to 0FFFFFH (excluding SFR and 2ndSFR)
	DSEG AT	0H to 0FFFFFH (excluding SFR and 2ndSFR)
	BSEG AT	0H to 0FFFFFH (excluding SFR and 2ndSFR)
	ORG	0H to 0FFFFFH (excluding SFR and 2ndSFR)
Symbol definition	EQU	20-bit value 0H to FFFFFH
	SET	20-bit value 0H to FFFFFH
Memory initialization and area reservation	DB	8-bit value 0H to FFH
	DW	16-bit value 0H to FFFFH
	DG	20-bit value 0H to FFFFFH
	DS	16-bit value 0H to FFFFH
Automatic branch instruction selection	BR/CALL	0H to FFFFFH

(2) Sizes of operands required by instructions

Instructions can be classified into machine instructions and directives. When they require immediate data or symbols, the size of the required operand differs according to the instruction or directive. An error occurs when source code specifies data that is larger than the required operand.

Expressions are operated as unsigned 32 bits. When evaluation results exceed 0FFFFFFFFH (32 bits),a warning message is issued.

However, when relocatable or external symbols are specified as operands, the value cannot be determined by the assembler. In these cases, the linker determines the value and performs range checks.

(3) Symbol attributes and relocation attributes of operands

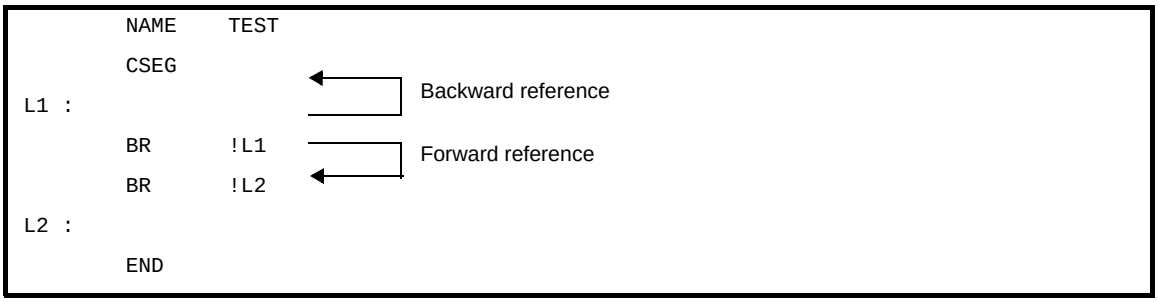
When names, labels, and $ (which indicate location counters) are described as instruction operands, they may or may not be describable as operands. This depends on the symbol attributes and relocation attributes (see "4.1.12 Restrictions on operations") .

When names and labels are described as instruction operands, they may or may not be describable as operands. This depends on the direction of reference.

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

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

<Example>



These symbol attributes and relocation attributes, as well as direction of reference for names and labels, are shown below.

Table 4-13. Properties of Described Symbols as Operands

	Symbol Attributes	NUMBER		ADDRESS				NUMBER ADDRESS		sfr Reserved Words[Note 1]	
	Relocation Attributes	Attributes Terms		Attributes Terms		Relocatable Terms		External Reference Terms			
	Reference Pattern	Backward	Forward	Backward	Forward	Backward	Forward	Backward	Forward	sfr	2ndsfr
Description Format	byte	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	word	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	saddr	OK	OK	OK	OK	OK	OK	OK	OK	OK[Note 3]	NG
	saddrp	OK	OK	OK	OK	OK	OK	OK	OK	OK[Note 2,4]	NG
	sfr	OK	OK	NG	NG	NG	NG	NG	NG	OK[Note 2,5]	NG
	sfrp	OK	OK	NG	NG	NG	NG	NG	NG	OK[Note 2,6]	NG
	addr20	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	addr16	OK	OK	OK	OK	OK	OK	OK	OK	OK[Note 7]	OK[Note 7]
	addr5	OK	OK	OK	OK	OK	OK	OK	OK	NG	NG
	bit	OK	OK	NG	NG	NG	NG	NG	NG	NG	NG
	n	OK	OK	OK	OK	NG	NG	NG	NG	NG	NG

Forward : Forward reference

Backward : Backward reference
OK : Description possible
NG : An error
- : Description impossible

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

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

3. sfr reserved word in saddr area

4. sfrp reserved word in saddr area

5. Only sfr reserved words that allow 8-bit accessing

6. Only sfr reserved words that allow 16-bit accessing

7. !sfr and !2ndsfr can be specified only for operand !addr16 of instructions other than BR and CALL.

Table 4-14. Properties of Described Symbols as Operands of Directives

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

Forward : Forward reference
Backward : Backward reference
OK : Description possible
- : Description impossible

Notes 1. Only an absolute expression can be described.

2. An error occurs if an expression including one of the following patterns is described.
- ADDRESS attribute - ADDRESS attribute
- ADDRESS attribute relational operator ADDRESS attribute

- HIGH absolute ADDRESS attribute
- LOW absolute ADDRESS attribute
- HIGHW absolute ADDRESS attribute
- LOWW absolute ADDRESS attribute
- DATAPOS absolute ADDRESS attribute
- MASK absolute ADDRESS attribute
- When the operation results can be affected by optimization from the above 8 patterns.

3. A term created by the HIGH/LOW/HIGHW/LOWW/DATAPOS/MASK operator that has a relocatable term is not allowed.
4. See "4.2.4 Memory initialization, area reservation directives".

4.2 Directives

This chapter explains the directives.

Directives are instructions that direct all types of instructions necessary for the 78K0R assemmbler to perform a series of processes.

4.2.1 Overview

Instructions are translated into object codes (machine language) as a result of assembling, but directives are not converted into object codes in principle.

Directives contain the following functions mainly:

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

The following table shows the types of directives.

Table 4-15. List of Directives

Type	Directives
Segment definition directives	CSEG, DSEG, BSEG, ORG
Symbol definition directives	EQU, SET
Memory initialization, area reservation directives	DB, DW, DG, DS, DBIT
Linkage directives	EXTRN, EXTBIT, PUBLIC
Object module name declaration directive	NAME
Branch instruction automatic selection directives	BR, CALL
Macro directives	MACRO, LOCAL, REPT, IRP, EXITM, ENDM
Assemble termination directive	END

The following sections explain the details of each directive.

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

4.2.2 Segment definition directives

The source module is described by dividing each segment unit.

The segment directive is what defines these "segments".

There are 4 types of these segments.

- Code segment
- Data segment
- Bit segment
- Absolute segment

The type of segment determines which area of the memory it is mapped to.

The following shows each segment definition method and the memory address that is mapped to.

Table 4-16. Segment Definition Method and Memory Address Location

Segment Type	Definition Method	Memory Address Location
Code segment	CSEG directive	In internal or external ROM address area
Data segment	DSEG directive	In internal or external RAM address area
Bit segment	BSEG directive	In internal RAM saddr area
Absolute segment	Specifies location address (AT location address) to relocation attribute with CSEG, DSEG, BSEG directive	Specified address

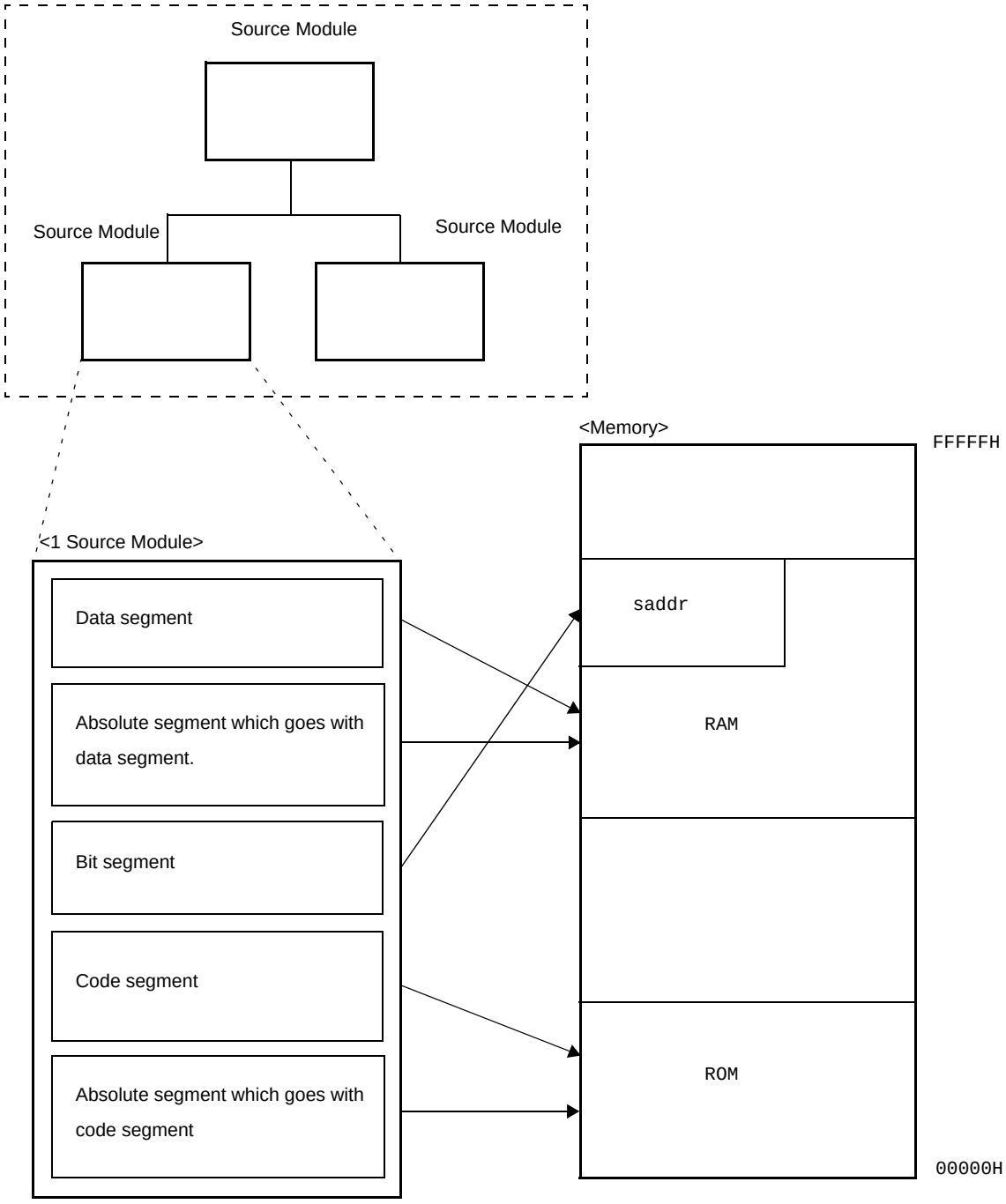
The absolute segment is defined for when the user wants to set the address mapped in the memory. For stack area, the user must set a stack pointer and secure an area in the data segment.

Also, segments cannot be located to the areas below.

Option byte area	C0 to C2H (user option byte) C3H (on-chip-debug option byte)
When specifying security ID	C4H to CDH
When using on-chip-debug function	02H to 03H, CE to D7H (for on-chip debugging) Area of program size part from the start address specified with the -go option by the user

Examples of segment mapping are shown below.

Figure 4-6. Segment Memory Mapping



The following segment definition directives are available.

Control Instruction	Overview
CSEG	Indicate to the assembler the start of a code segment
DSEG	Indicate to the assembler the start of a data segment
BSEG	Indicate to the assembler the start of a bit segment
ORG	Set the value of the expression specified by its operand of the location counter.

CSEG

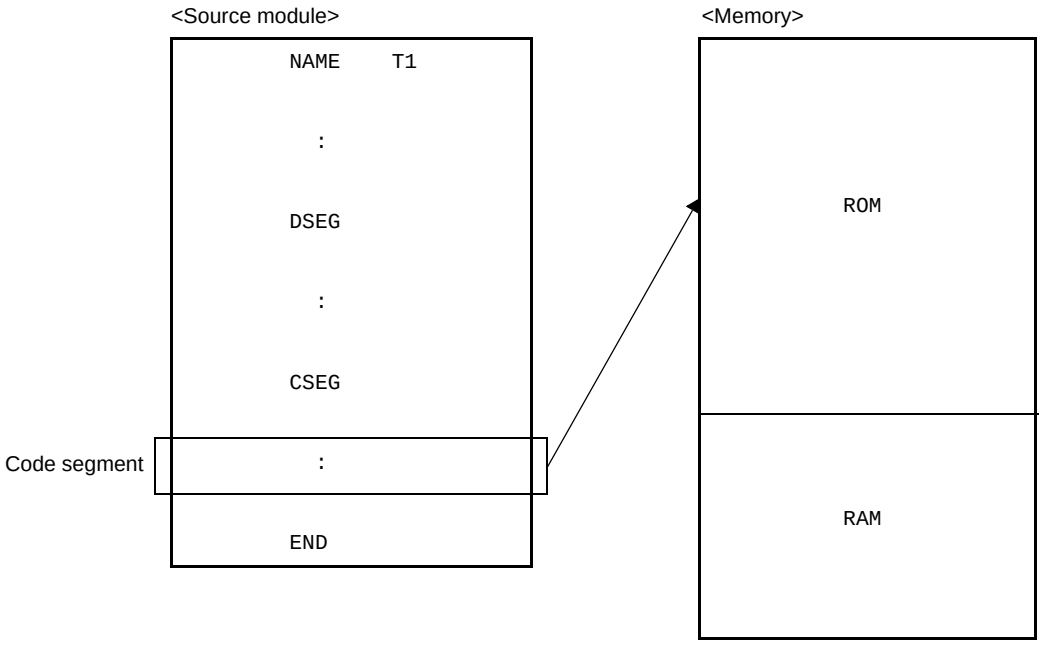
Indicate to the assembler the start of a code segment.

[Description Format]

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

[Function]

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



[Use]

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

[Description]

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

Table 4-17. Relocation Attributes of CSEG

Relocation Attribute	Description Format	Explanation
CALLT0	CALLT0	Tells the assembler to locate the specified segment so that the start address of the segment becomes a multiple of 2 within the address range 00080H to 000BFH.
FIXED	FIXED	Tells the assembler to locate the beginning of the specified segment within the address range 000C0H to 0FFFFH
BASE	BASE	Tells the assembler to locate the beginning of the specified segment within the address range 000C0H to 0FFFFH
AT	AT *absolute-expression*	Tells the assembler to locate the specified segment to an absolute address (excluding SFR and 2ndSFR).
UNIT	UNIT	Tells the assembler to locate the specified segment to any address (000C0H to EFFFFH in memory area "ROM").
UNITP	UNITP	Tells the assembler to locate the specified segment to any address, so that the start of the address may be an even number (000C0H to EFFFFH) in memory area "ROM").
IXRAM	IXRAM	Tells the assembler to locate the specified segment to any address (000C0H to EFFFFH in memory area "ROM").
SECUR_ID	SECUR_ID	It is a security ID specific attribute. Not specify except security ID. Tells the assembler to locate the specified segment within the address range 000C4H to 000CDH.
PAGE64KP	PAGE64KP	Tells the assembler to locates the specified segment in memory area "ROM" that does not extend over a 64 KB boundary, so that the start of the address may be an even number. The same-named segments but located in different files are not combined.
UNIT64KP	UNIT64KP	Tells the assembler to locates the specified segment in memory area "ROM" that does not extend over a 64 KB boundary, so that the start of the address may be an even number. The same-named segments are combined.
MIRRORP	MIRRORP	Tells the assembler to locates the specified segment in the area mirrored in the RAM space when MAA = 0 (01000H to 0xxxxH) or the area mirrored in the RAM space when MAA = 1 (11000H to 1xxxxH).[Note]
OPT_BYTE	OPT_BYTE	It is a user option byte and on-chip debugging specific attribute. Not specify except user option byte and on-chip debugging. Tells the assembler to locate the specified segment within the address range 000C0H to 000C3H.

Note The address ranges to be mirrored in the RAM space differ depending on the device used.

- If no relocation attribute is specified for the code segment, the assembler will assume that "UNIT" has been specified.
- If a relocation attribute other than those listed in "Table 4-17. Relocation Attributes of CSEG" is specified, the assembler will output an error and assume that "UNIT" has been specified. An error occurs if the size of each code segment exceeds that of the area specified by its relocation attribute.
- If the absolute expression specified with the relocation attribute "AT" is illegal, the assembler will output an error message and continue processing by assuming the value of the expression to be "0".

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

Relocation Attribute	Default Segment Name
CALLT0	?CSEGT0
FIXED	?CSEGFX
UNIT (or omitted)	?CSEG
UNITP	?CSEGUP
IXRAM	?CSEGIX
BASE	?CSEGB
SECUR_ID	?CSEGSI
PAGE64KP	?CSEGP64
UNIT64KP	?CSEGU64
MIRRORP	?CSEGMIP
OPT_BYTE	?CSEGOB0
AT	Segment name cannot be omitted.

- When the size of the following segment is 0 among the default segments that C compiler outputs, the relocation attribute is changed by the linker.

Section Name	Relocation Attribute	Relocation Attribute When Being Size 0
@@CALT	CSEG CALLT0	CSEG UNIT
@@CNST	CSEG MIRRORP	CSEG UNIT

- An error occurs if the segment name is omitted when the relocation attribute is AT.
- If two or more code segments have the same relocation attribute (except AT), these code segments may have the same segment name.
  These same-named code segments are processed as a single code segment within the assembler.
  An error occurs if the same-named segments differ in their relocation attributes. Therefore, the number of the same-named segments for each relocation attribute is one.
- Description of a code segment can be divided into units. The same relocation attribute and the samenamed code segment described in one module are handled by the assembler as a series of segments.

Cautions 1. Description of a code segment whose relocation attribute is AT cannot be divided into units.

2. Insert a 1-byte interval, as necessary, so that the address specified by relocation attribute CALLT0 may be an even number.

- The same-named data segments in two or more different modules can be specified only when their relocation attributes are UNIT, CALLT0, FIXED, UNITP, BASE, PAGE64KP, UNIT64KP, MIRRORP, or SECUR_ID, and are combined into a single data segment at linkage.
- No segment name can be referenced as a symbol.
- The total number of segments that can be output by the assembler is up to 256 alias names, including those defined with the ORG directive. The same-named segments are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

- Specify user option byte and on-chip debugging by using OPT_BYTE.
  An error occurs if the option byte is specified for the chip without the option byte feature.
  When the user option byte is not specified for the chip having the user option byte feature, define a default segment of “?CSEGOB0” to each address and set the initial value by reading from a device file.

[Example]

```
        NAME    SAMP1
C1      CSEG            ; (1)

C2      CSEG    CALLT0  ; (2)

        CSEG    FIXED   ; (3)

C1      CSEG    CALLT0  ; (4)   <- Error

        CSEG            ; (5)
        END
```

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

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

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

(4) An error occurs because the segment name "C1" was defined as the relocation attribute "UNIT" in (1).

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

DSEG

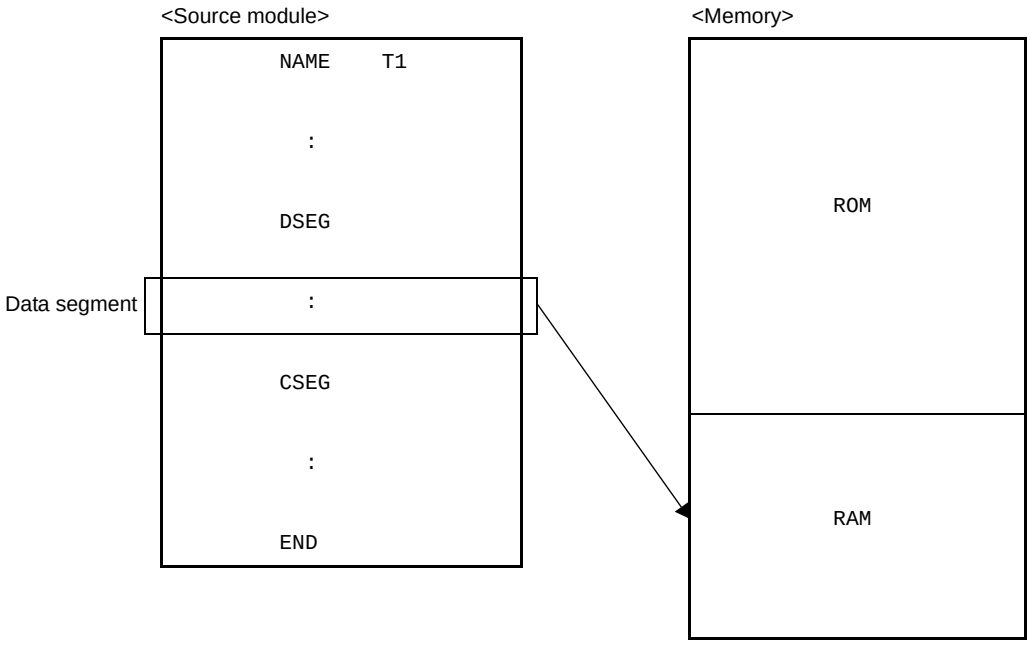
Indicate to the assembler the start of a data segment.

[Description Format]

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

[Function]

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



[Use]

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

Value Specified by -w	Check Target
0	No areas
1	RB0
2	RB0 to RB3

[Description]

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

Table 4-18. Relocation Attributes of DSEG

Relocation Attribute	Description Format	Explanation
SADDR	SADDR	Tells the assembler to locate the specified segment in the saddr area (saddr area: FFE20H to FFEFFH).
SADDRP	SADDRP	Tells the assembler to locate the specified segment from an even-numbered address of the saddr area (saddr area: FFE20H to FFEFFH).
AT	AT *absolute-expression*	Tells the assembler to locate the specified segment in an absolute address (excluding SFR and 2ndSFR).
UNIT	UNIT or no specification	Tells the assembler to locate the specified segment in the internal or any external location (within the memory area name "RAM").
UNITP	UNITP	Tells the assembler to locate the specified segment in the internal or any external location from an even-numbered address (within the memory area name "RAM").
BASEP	BASEP	Tells the assembler to locates the specified segment in the internal RAM area so that the start of the address may be an even number (not including saddr area: FxxxxH to FFEFFH).[Note] When arranging the data to access without ES references, it's used.
PAGE64KP	PAGE64KP	Tells the assembler to locates the specified segment in memory area "RAM" that does not extend over a 64 KB boundary, so that the start of the address may be an even number. The same-named segments but located in different files are not combined.
UNIT64KP	UNIT64KP	Tells the assembler to locates the specified segment in memory area "RAM" that does not extend over a 64 KB boundary, so that the start of the address may be an even number. The same-named segments are combined.

Note The address represented by xxxx varies depending on the device used.

- Relocation attributes provided for the 78K0 assembler can also be described, which function in the same manner as "UNIT".
  The following table lists the relocation attributes of DSEG provided for the 78K0.

Relocation aAtribute	Description ffrmat
IHRAM	IHRAM
LRAM	LRAM
DSPRAM	DSPRAM
IXRAM	IXRAM

- If no relocation attribute is specified for the data segment, the assembler will assume that "UNIT" has been specified.

- If a relocation attribute other than those listed in "Table 4-18. Relocation Attributes of DSEG" is specified, the assembler will output an error and assume that "UNIT" has been specified. An error occurs if the size of each data segment exceeds that of the area specified by its relocation attribute.
- If the absolute expression specified with the relocation attribute "AT" is illegal, the assembler will output an error and continue processing by assuming the value of the expression to be "0".
- Machine language instructions (including BR directive) cannot be described in a data segment. If described, an error is output and the line is ignored.
- By describing a segment name in the symbol field of the DSEG directive, the data segment can be named. If no segment name is specified for a data segment, the assembler automatically gives a default segment name. The default segment names of the data segments are shown below.

Relocation Atribute	Default Segment Name
SADDR	?DSEGS
SADDRP	?DSEGSP
UNIT(or no specification)	?DSEG
UNITP	?DSEGUP
IHRAM	?DSEGIH
LRAM	?DSEGL
DSPRAM	?DSEGDSP
IXRAM	?DSEGIX
BASEP	?DSEGBP
PAGE64KP	?DSEGP64
UNIT64KP	?DSEGU64
AT	Segment name cannot be omitted.

- When the size of the following segment is 0 among the default segments that C compiler outputs, the relocation attribute is changed by the linker.

Section Name	Relocation Attribute	Relocation Attribute When Being Size 0
@@INIS	DSEG SADDRP	DSEG UNITP
@@DATS	DSEG SADDRP	DSEG UNITP
@EINIS	DSEG SADDRP	DSEG UNITP
@EDATS	DSEG SADDRP	DSEG UNITP

- If two or more data segments have the same relocation attribute (except AT), these data segments may have the same segment name.
  These segments are processed as a single data segment within the assembler.
- Description of a data segment can be divided into units. The same relocation attribute and the same-named code segment described in one module are handled by the assembler as a series of segments.

Cautions 1. Description of a code segment whose relocation attribute is AT cannot be divided into units.

2. When the relocation attribute is SADDR, insert a 1-byte interval, as necessary, so that the address immediately after a DESG directive is described may be an even number.

- If the relocation attribute is SADDRP, the specified segment is located so that the address immediately after the DSEG directive is described becomes a multiple of 2.

- An error occurs if the same-named segments differ in their relocation attributes. Therefore, the number of the same-named segments for each relocation attribute is one.
- The same-named data segments in two or more different modules can be specified only when their relocation attributes are UNIT, UNITP, SADDR, SADDRP, LRAM, IHRAM, DSPRAM, IXRAM, BASEP, PAGE64KP, or UNIT64KP, and are combined into a single data segment at linkage.
- No segment name can be referenced as a symbol.
- The total number of segments that can be output by the assembler is up to 255 alias segments including those defined with the ORG directive. The same-named segments are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

[Example]

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

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

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

(3) This description corresponds to "MOV A, saddr".
Relocatable label "WORK2" cannot be described as "saddr". Therefore, an error occurs as a result of this description.

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

(5) This description corresponds to "MOVW AX, saddrp".
Relocatable label "WORK1" cannot be described as "saddrp". Therefore, an error occurs as a result of this description.

BSEG

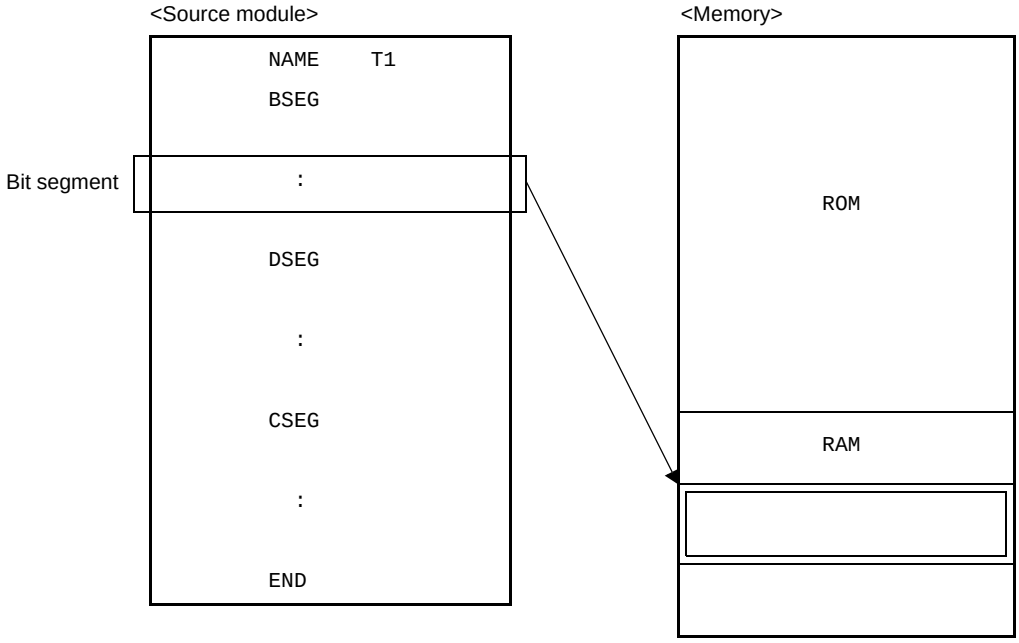
Indicate to the assembler the start of a bit segment.

[Description Format]

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

[Function]

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



[Use]

- Describe the DBIT directive in the bit segment defined by the BSEG directive.
- No instructions can be described in any bit segment.

[Description]

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

Table 4-19. Relocation Attributes of BSEG

Relocation Attribute	Description Format	Explanation
AT	AT *absolute-expression*	Tells the assembler to locate the starting address of the specified segment in the 0th bit of an absolute address. Specification in bit units is prohibited (00000H to FFFFFH)(excluding SFR and 2ndSFR).
UNIT	UNIT or no specification	Tells the assembler to locate the specified segment in any location (FFE20H to FFEFFH).

- If no relocation attribute is specified for the bit segment, the assembler assumes that "UNIT" is specified.
- If a relocation attribute other than those listed in Table 3-5 is specified, the assembler outputs an error and assumes that "UNIT" is specified. An error occurs if the size of each bit segment exceeds that of the area specified by its relocation attribute.
- In both the assembler and the linker, the location counter in a bit segment is displayed in the form “0xxxxx.b” (The byte address is hexadecimal 5 digits and the bit position is hexadecimal 1 digit (0 to 7)).

(1) Absolute

Byte Address	Bit Position							
	0	1	2	3	4	5	6	7
0FFE20H	0FFE20H.0	0FFE20H.1	0FFE20H.2	0FFE20H.3	0FFE20H.4	0FFE20H.5	0FFE20H.6	0FFE20H.7
0FFE21H	0FFE21H.0	0FFE21H.1	0FFE21H.2	0FFE21H.3	0FFE21H.4	0FFE21H.5	0FFE21H.6	0FFE21H.7

(2) Relocatable

Byte Address	Bit Position							
	0	1	2	3	4	5	6	7
0H	0H.0	0H.1	0H.2	0H.3	0H.4	0H.5	0H.6	0H.7
1H	1H.0	1H.1	1H.2	1H.3	1H.4	1H.5	1H.6	1H.7

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

Symbol Value	Bit Offset
0FFE20H.0	0000
0FFE20H.1	0001
0FFE20H.2	0002
:	:
0FFE20H.7	0007
0FFE21H.0	0008
0FFE21H.1	0009
:	:
0FFE80H.0	0300

Symbol Value	Bit Offset
:	:

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

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

- When the size of the following segment is 0 among the default segments that C compiler outputs, the relocation attribute is changed by the linker.

Section Name	Relocation Attribute	Relocation Attribute When Being Size 0
@@BITS	BSEG UNIT (in SADDR area)	BSEG UNIT (in RAM area)

- If the relocation attribute is "UNIT", two or more data segments can have the same segment name (except AT). These segments are processed as a single segment within the assembler. Therefore, the number of same-named segments for each relocation attribute is one.
- The same-named bit segments name must have the same relocation attribute UNIT (when the relocation attribute is AT, specifying the same name for multiple segments is prohibited).
- If the relocation attribute of the same-named segments in a module is not UNIT, an error is output and the line is ignored.
- The same-named bit segments in two or more different modules will be combined into a single bit segment at linkage time.
- No segment name can be referenced as a symbol.
- Bit segments are located at 0H to FFFFFH by the linker.
- Labels cannot be described in a bit segment.
- The only instructions that can be described in the bit segments are the DBIT, EQU, SET, PUBLIC, EXTBIT, EXTRN, MACRO, REPT, IRP, ENDM directive, macro definition and macro reference. Description of instructions other than these causes in an error.
- The total number of segments that the assembler outputs is up to 256 alias segments, with segments defined by the ORG directive. The segments having the same name are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

[Example]

```
        NAME    SAMP1

FLAG    EQU     0FFE20H
FLAG0   EQU     FLAG.0          ; (1)
FLAG1   EQU     FLAG.1          ; (2)

        BSEG                    ; (3)
FLAG2   DBIT

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

(1) Bit addresses (bits 0 of 0FFE20H are defined with consideration given to byte address boundaries.

(2) Bit addresses (bits 1 of 0FFE20H) are defined with consideration given to byte address boundaries.

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

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

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

ORG

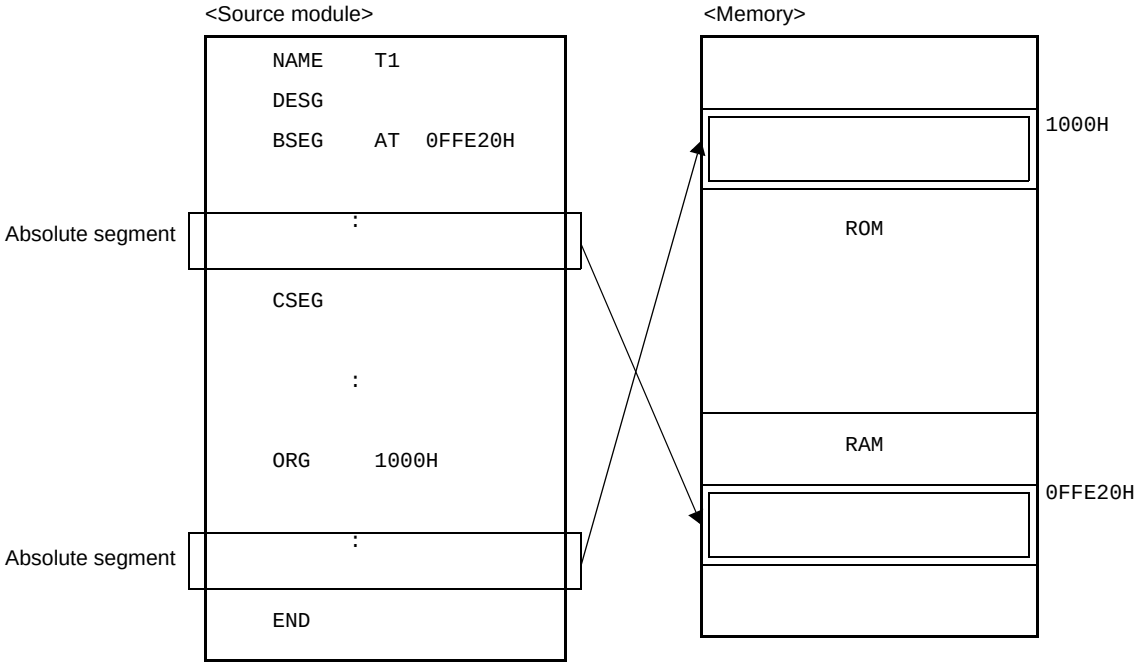
Set the value of the expression specified by its operand of the location counter.

[Description Format]

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

[Function]

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



[Use]

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

[Description]

- The absolute segment defined with the ORG directive belongs to the code segment or data segment defined with the CSEG or DSEG directive immediately before this ORG directive.
  Within an absolute segment that belongs to a data segment, no instructions can be described. An absolute segment that belongs to a bit segment cannot be described with the ORG directive.
- The code segment or data segment defined with the ORG directive is interpreted as a code segment or data segment of the relocation attribute "AT".
- By describing a segment name in the symbol field of the ORG directive, the absolute segment can be named. The maximum number of characters that can be recognized as a segment name is 8.
- The same-named segments in a module, which are defined with the ORG directive, are handled in the same manner as segments of the AT attribute, which are defined with the CSEG or DESG directive.

- The same-named segments in different modules, which are defined with the ORG directive, are handled in the same manner as segments of the AT attribute, which are defined with the CSEG or DESG directive.
- If no segment name is specified for an absolute segment, the assembler will automatically assign the default segment name "?A0nnnnn", where "nnnnn" indicates the 5 digit hexadecimal start address (00000 to FFFFF) of the segment specified.
- If neither CSEG nor DSEG directive has been described before the ORG directive, the absolute segment defined by the ORG directive is interpreted as an absolute segment in a code segment.
- If a name or label is described as the operand of the ORG directive, the name or label must be an absolute term that has already been defined in the source module.
- If illegal objects are described for absolute expressions, or if the evaluated value of an absolute expression exceeds 00000H to FFEFFH, the assembler outputs an error and continues processing, assuming that the value of the absolute expression is 00000H.
- Absolute expressions for operands are evaluated in unsigned 32-bit units.
- No segment name can be referenced as a symbol.
- The total number of segments that the assembler outputs is up to 256 alias segments, with segments defined by the segment definition directives. The segments having the same name are counted as one.
- The maximum number of characters recognizable as a segment name is 8.
- The uppercase and lowercase characters of a segment name are distinguished.

[Example]

```
        NAME    SAMP1

        DSEG
        ORG     0FFE20H         ; (1)
SADR1 : DS      1
SADR2 : DS      1
SADR3 : DS      2

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

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

(1) An absolute segment that belongs to a data segment is defined.
This absolute segment will be located from the short direct addressing area that starts from address "FFE20H". Because specification of the segment name is omitted, the assembler automatically assigns the name "?A0FFE20".

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

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

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

4.2.3 Symbol definition directives

Symbol definition directives specify names for the data that is used when writing to source modules. With these, the data value specifications are made clear and the details of the source module are easier to understand.

Symbol definition directives indicate the names of values used in the source module to the assembler.

The following symbol definition directives are available.

Control Instruction	Overview
EQU	The value of the expression specified by operand and the numerical data with attribute are defined as a name.
SET	The value of the expression specified by operand and the variable with attribute are defined as a name.

EQU

The value of the expression specified by operand and the numerical data with attribute are defined as a name.

[Description Format]

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

[Function]

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

[Use]

- Define numerical data to be used in the source module as a name with the EQU directive and describe the name in the operand of an instruction in place of the numerical data.
  Numerical data to be frequently used in the source module is recommended to be defined as a name. If you must change a data value in the source module, all you need to do is to change the operand value of the name.

[Description]

- The EQU directive may be described anywhere in a source program.
- A symbol defined with the EQU directive cannot be redefined with the SET directive, nor as a label. In addition, a symbol or label defined with the SET directive cannot be redefined with the EQU directive, nor as a label.
- When a name or label is to be described in the operand of the EQU directive, use the name or label that has already been defined in the source module.
  No external reference term can be described as the operand of this directive.
  SFRs and SFR bit symbols can be described.
- An expression including a term created by a HIGH/LOW/HIGHW/LOWW/DATAPOS/BITPOS operator that has a relocatable term in its operand cannot be described.
- An error occurs if an expression with any of the following patterns of operands is described:

(1) Expression 1 with ADDRESS attribute - Expression 2 with ADDRESS attribute
Either of the following conditions (1) and (2) is fulfilled in the above expression (a) or (b):

(a) (a) If label 1 in the expression 1 with ADDRESS attribute and label 2 in the expression 2 with ADDRESS attribute belong to the same segment and if a BR directive for which the number of bytes of the object code cannot be determined is described between the two labels

(b) (b) If label 1 and label 2 differ in segment and if a BR directive for which the number of bytes of the object code cannot be determined is described between the beginning of the segment and label

(2) Expression 1 with ADDRESS attribute attributeRelational operator Expression 2 with ADDRESS attribute

(3) HIGH absolute expression with ADDRESS attribute

(4) LOW absolute expression with ADDRESS attribute

(5) HIGHW absolute expression with ADDRESS attribute

(6) LOWW absolute expression with ADDRESS attribute

(7) DATAPOS absolute expression with ADDRESS attribute

(8) BITPOS absolute expression with ADDRESS attribute

(9) The following (a) is fulfilled in the expression (3) to (8):

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

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

Operand Type	Symbol Value	Reference Range
A.bit[Note 1]	1.bit	Can be referenced within the same module only.
PSW.bitye	0FFFFAH.bit	
sfr[Note 2].bit[Note 1]	0FFFXXH[Note 3].bit	
2ndsfr[Note 2].bit[Note 1]	0FXXXXH[Note 4].bit	
saddr.bit[Note 1]	0FFXXXH[Note 5].bit	Can be referenced from another module.
expression.bit[Note 1]	0XXXXXH[Note 6].bit	

Notes 1. bit = 0 to 7

2. For a detailed description, see the user's manual of each device.

3. 0FFFXXH : the address of an sfr

4. 0FXXXXH : 2ndsfr area

5. 0FXXXXH : saddr area (0FFE20H to 0FFF1FH)

6. 0XXXXXH : 0H to 0FFFFFH

[Example]

```
        NAME    SAMP1

WORK1   EQU     0FFE20H         ; (1)
WORK10  EQU     WORK1.0         ; (2)
P02     EQU     P0.2            ; (3)
A4      EQU     A.4             ; (4)
PSW5    EQU     PSW.5           ; (5)

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

(1) The name "WORK1" has the value "0FFE20H", symbol attribute "NUMBER", and relocation attribute "ABSOLUTE".

(2) The name "WORK10" is assigned to bit value "WORK1.0", which is in the operand format "saddr.bit". "WORK1", which is described in an operand, is already defined at the value “0FFE20H”, in (1) above.

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

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

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

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

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

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

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

Names that have defined "A.bit", and "PSW.bit" as in (4) through (5) can be referenced only within the same module.

A name that has defined "sfr.bit", "saddr.bit", "*expression*.bit" can also be referenced from another module as an external definition symbol (see "4.2.5 Linkage directives").

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

```
        Assemble list

ALNO    STNO    ADRS    OBJECT      M I SOURCE STATEMENT

  1       1                                 NAME    SAMP
  2       2
  3       3           ( FFE20 )       WORK1   EQU     0FFE20H         ; (1)
  4       4           ( FFE20.0 )     WORK10  EQU     WORK1.0         ; (2)
  5       5           ( FFF00.2 )     P02     EQU     P0.2            ; (3)
  6       6           ( 00001.4 )     A4      EQU     A.4             ; (4)
  7       7           ( FFFFA.5 )     PSW5    EQU     PSW.5           ; (5)
  8       8
  9       9   00000   710220                  SET1    WORK10          ; (6)
 10      10   00003   712200                  SET1    P02             ; (7)
 11      11   00006   71CA                    SET1    A4              ; (8)
 12      12   00008   715AFA                  SET1    PSW5            ; (9)
 13      13                                    END
```

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

SET

The value of the expression specified by operand and the variable with attribute are defined as a name.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

[Example]

```
        NAME    SAMP1
COUNT   SET     10H             ; (1)

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

COUNT   SET     20H             ; (3)

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

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

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

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

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

4.2.4 Memory initialization, area reservation directives

The memory initialization directive defines the constant data used by the program.

The defined data value is generated as object code.

The area reservation directive secures the area for memory used by the program.

The following memory initialization and partitioning directives are available.

Control Instruction	Overview
DB	Initialization of byte area
DW	Initialization of word area
DG	Initialization of 20 bit area in 32 bits (4 bytes)
DS	Secures the memory area of the number of bytes specified by operand.
DBIT	Secures 1 bit of memory area in bit segment.

DB

Initialization of byte area

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified. Otherwise, an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler initializes an area equivalent to the specified number of bytes with the value "00H".

(b) An absolute expression must be described as a size. If the size description is illegal, the assembler outputs an error message and will not execute initialization.

(2) With initial value specification:

(a) Expression

The value of an expression must be 8-bit data. Therefore, the value of the operand must be in the range of 0H to 0FFH. If the value exceeds 8 bits, the assembler will use only lower 8 bits of the value as valid data and output an error.

(b) Character string

If a character string is described as the operand, an 8-bit ASCII code will be reserved for each character in the string.

- The DB directive cannot be described in a bit segment.
- Two or more initial values may be specified within a statement line of the DB directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.

[Example]

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

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

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

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

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

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

DW

Initialization of word area

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified; otherwise an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler will initialize an area equivalent to the specified number of words with the value "00H".

(b) An absolute expression must be described as a size. If the size description is illegal, the assembler outputs an error and will not execute initialization.

(2) With initial value specification:

(a) Constant
16 bits or less.

(b) Expression
The value of an expression must be stored as a 16-bit data.
No character string can be described as an initial value.

- The DW directive cannot be described in a bit segment.
- The upper 2 digits of the specified initial value are stored in the HIGH address and the lower 2 digits of the value in the LOW address.
- Two or more initial values may be specified within a statement line of the DW directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.

[Example]

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

                CSEG
MAIN :
                CSEG
SUB1 :

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

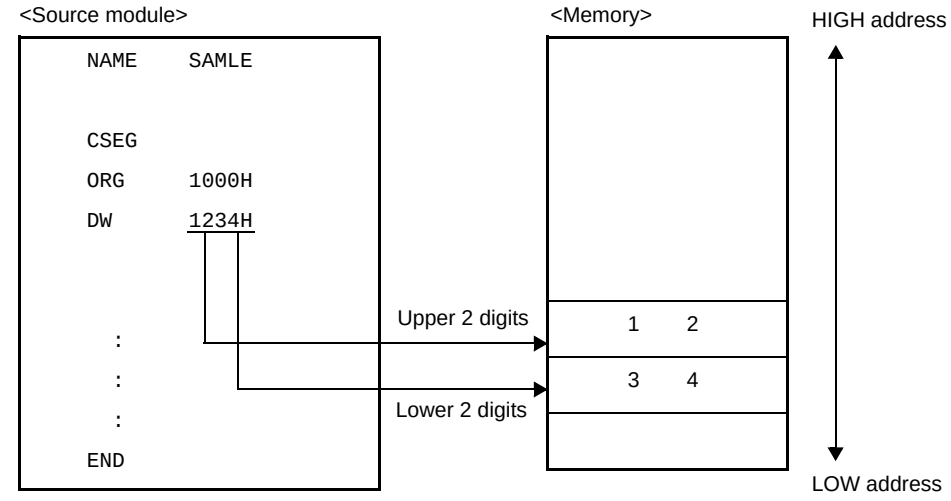
(1) Because the size is specified, the assembler will initialize each word with the value "00H".

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

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

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

<Example>



DG

Initialization of 20 bit area in 32 bits (4 bytes)

[Description Format]

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

[Function]

- The DG directive tells the assembler to initialize a 20-bit area in 32-bit (4-byte) units. The initial value or size can be specified as an operand.
- The DG directive also tells the assembler to initialize a memory area in 4 bytes units with the initial value(s) specified in the operand field.

[Use]

- Use the DG directive when defining a 20-bit numeric constant such as an address or data used in the program.

[Description]

- If a value in the operand field is parenthesized, the assembler assumes that a size is specified; otherwise an initial value is assumed.

(1) With size specification:

(a) If a size is specified in the operand field, the assembler will initialize an area equivalent to the specified numbers x 4 bytes, with "00H".

(b) An absolute expression must be described as a size. If the size description is illegal, the assembler outputs an error and will not execute initialization.

(2) With initial value specification:

(a) Constant
20 bits or less.

(b) Expression
The value of an expression must be stored as a 16-bit data.
No character string can be described as an initial value.

- The DG directive cannot be described in a bit segment.
- The highest byte of the specified initial value is stored in the HIGH WORD address, the lowest byte is stored in the LOW address, and the higher byte of the lowest 2 bytes is stored in the HIGH address in the memory.
- Two or more initial values may be specified within a statement line of the DW directive.
- As an initial value, an expression that includes a relocatable symbol or external reference symbol may be described.

[Example]

```
                NAME    SAMP1

DATA1 :         DG      12345H, 56789H          ; (1)
DATA2 :         DG      ( 10 )                  ; (2)
                END
```

(1) A 4-byte area is initialized with value "4523010089670500".

(2) The 40-byte (10 x 4 bytes) area is initialized with "00H".

Caution For the 20-bit value, the HIGH WORD address in the memory is initialized with the highest byte, the LOW address in the memory is initialized with the lowest byte, and the HIGH address is initialized with the higher byte of the lowest 2 bytes.

<Example>

<Source module>

```
NAME    SAMP1
        CSEG
DATA1:  DG      12345H, 56789H
          :
        END
```

<Memory>

HW	00
H	05
L	67
	89
HW	00
H	01
L	23
	45

HIGH Address

LOW address

HW : HIGH WORD

H : HIGH

L : LOW

DS

Secures the memory area of the number of bytes specified by operand.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

[Example]

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

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

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

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

DBIT

Secures 1 bit of memory area in bit segment.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The DBIT directive is described only in a bit segment.
- The contents of a 1-bit area reserved with the DBIT directive are unknown (indefinite).
- If a name is specified in the Symbol field, the name has an address and a bit position as its value.
- The defined name can be described at the place where saddr.bit, addr16.bit, ES:addr16.bitt is required.

[Example]

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

        CSEG
        SET1    BIT1    ; (2)

        CLR1    BIT2    ; (3)
        END
```

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

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

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

4.2.5 Linkage directives

Linkage directives clarify associations when referring to symbols defined by other modules. This is thought to be in cases when one program is written that divides module 1 and module 2.

In cases when you want to refer to a symbol defined in module 2 in module 1, there is nothing declared in either module and and so the symbol cannot be used. Due to this, there is a need to display "I want to use" or "I don't want to use" in respective modules.

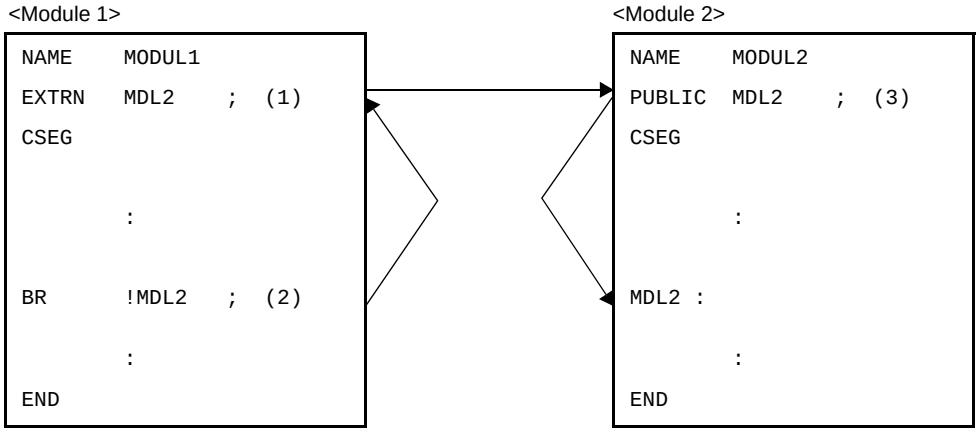
An "I want to refer to a symbol defined in another module" external reference declaration is made in module 1. At the same time, a "This symbol may be referred to by other symbols" external definition declaration is made in module 2.

This symbol can only begin to be referred to after both external reference and external definition declarations in effect.

Linkage directives are used to to form this relationship and the following instructions are available.

- Symbol external reference declaration: EXTRN, and also EXTBIT directive.
- Symbol external definition declaration: PUBLIC directive.

Figure 4-7. Relationship of Symbols Between 2 Modules



In the above modules, in order for the "MDL2" symbol defined in module 2 to be referred to in (2), an external reference is declared via an EXTRN directive in (1).

In module 2 (3), an external definition declaration is undergone of the "MDL2" symbol referenced from module 1 via a PUBLIC directive.

Whether or not this external reference and external definition symbols are correctly responding or not is checked via a linker.

The following linkage directives are available.

Control Instruction	Overview
EXTRN	Declares to the linker that a symbol (other than bit symbols) in another module is to be referenced in this module.
EXTBIT	Directive declares to the linker that a bit symbol in another module is to be referenced in this module.
PUBLIC	Declares to the linker that the symbol described in the operand field is a symbol to be referenced from another module.

EXTRN

Declares to the linker that a symbol (other than bit symbols) in another module is to be referenced in this module.

[Description Format]

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	EXTRN	*symbol-name*[, ...]	[; comment]
		or	
[*label*:]	EXTRN	BASE(*symbol-name*[, ...])	[; comment]

[Function]

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

[Use]

- When referencing a symbol defined in another module, the EXTRN directive must be used to declare the symbol as an external reference.
- The resulting operation varies depending on the description format for operands.

BASE(symbol-name[, ...])	The specified symbol is regarded as a symbol in an area within a 64 KB area (0H to 0FFFF) and can be referenced.
No relocation attribute specified	After located by the linker, processing is performed in accordance with the area for which PUBLIC is declared and then can be referenced.

[Description]

- The EXTRN directive may be described anywhere in a source program (see "4.1.1 Basic configuration").
- Up to 20 symbols can be specified in the operand field by delimiting each symbol name with a comma (,).
- When referencing a symbol having a bit value, the symbol must be declared as an external reference with the EXTBIT directive.
- The symbol declared with the EXTRN directive must be declared in another module with a PUBLIC directive.
- No error is output even if a symbol declared with the EXTRN directive is not referenced in the module.
- No macro name can be described as the operand of EXTRN directive (see "4.4 Macros" for the macro name).
- The EXTRN directive enables only one EXTRN declaration for a symbol in an entire module. For the second and subsequent EXTRN declarations for the symbol, the linker will output a warning.
- A symbol that has been declared cannot be described as the operand of the EXTRN directive. Conversely, a symbol that has been declared as EXTRN cannot be redefined or declared with any other directive.
- An area within a 64 KB area (0H to 0FFFFH) can be referenced using a symbol defined with the EXTRN directive. A symbol name declared in the format of "BASE(symbol name)" can be referenced from the 64 KB area.

[Example]

- Module 1

```
        NAME    SAMP1
        EXTRN   SYM1, SYM2, BASE (SYM3) ; (1)
        CSEG
S1 :    DW      SYM1                    ; (2)
        MOV     A, SYM2                 ; (3)
        BR      !SYM3                   ; (4)
        END
```

- Module 2

```
        NAME    SAMP2
        PUBLIC  SYM1, SYM2, SYM3        ; (5)

        CSEG
SYM1    EQU     0FFH                    ; (6)
DATA1   DSEG    SADDR
SYM2 :  DB      012H                    ; (7)
C1      CSEG    BASE
SYM3 :  MOV     A, #20H                 ; (8)
        END
```

(1) This EXTRN directive declares symbols "SYM1", "SYM2" and "SYM3" to be referenced in (2), (3) and (4) as external references. Two or more symbols may be described in the operand field.

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

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

(4) This BR instruction references symbol "SYM3" and outputs a code that references an area within a 64 KB area (0H to 0FFFFH).

(5) The symbols "SYM1", "SYM2" and "SYM3" are declared as external definitions.

(6) The symbol "SYM1" is defined.

(7) The symbol "SYM2" is defined.

(8) The symbol "SYM3" is defined.

EXTBIT

Directive declares to the linker that a bit symbol in another module is to be referenced in this module.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The EXTBIT directive may be described anywhere in a source program.
- Up to 20 symbols can be specified in the operand field by delimiting each symbol with a comma (,).
- A symbol declared with the EXTBIT directive must be declared with a PUBLIC directive in another module.
- The EXTBIT directive enables only one EXTBIT declaration for a symbol in an entire module. For the second and subsequent EXTBIT declarations for the symbol, the linker will output a warning.
- No error is output even if a symbol declared with the EXTRN directive is not referenced in the module.

[Example]

- Module 1

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

- Module 2

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

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

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

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

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

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

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

PUBLIC

Declares to the linker that the symbol described in the operand field is a symbol to be referenced from another module.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The PUBLIC directive may be described anywhere in a source program.
- Up to 20 symbols can be specified in the operand field by delimiting each symbol name with a comma (,).
- Symbol(s) to be described in the operand field must be defined within the same module.
- The PUBLIC directive enables only one PUBLIC declaration for a symbol in an entire module. The second and subsequent PUBLIC declarations for the symbol will be ignored by the linker.
- Bit symbols in each bit area can be declared with PUBLIC.
- The following symbols cannot be used as the operand of the PUBLIC directive:

(1) Name defined with the SET directive

(2) Symbol defined with the EXTRN or EXTBIT directive within the same module

(3) Segment name

(4) Module name

(5) Macro name

(6) Symbol not defined within the module

(7) Symbol defining an operand with a SFBIT attribute with the EQU directive

(8) Symbol defining an sfr and 2ndSFR with the EQU directive (however, the place where sfr area and saddr area are overlapped is excluded)

[Example]

- Module 1

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

A1      EQU     10H
A2      EQU     0FFE20H.1

        CSEG
        BR      B1
        SET1    C1
        END
```

- Module 2

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

- Module 3

```
        NAME    SAMP3
        PUBLIC  C1              ; (3)
        EXTBIT  A2
C1      EQU     0FFE21H.0
        CSEG
        CLR1    A2
        END
```

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

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

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

4.2.6 Object module name declaration directive

An object module name directive gives a name to an object module generated by the assembler.
The following object module name declaration directives are available.

Control Instruction	Overview
NAME	Assign the object module name described in the operand field to an object module to be output by the assembler.

NAME

Assign the object module name described in the operand field to an object module to be output by the assembler.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The NAME directive may be described anywhere in a source program.
- For the conventions of module name description, see the conventions on symbol description in "(3) Symbol field".
- Characters that can be specified as a module name are those characters permitted by the operating system of the assembler software other than "(" (28H) or ")" (29H) or 2-byte characters.
- No module name can be described as the operand of any directive other than NAME or of any instruction.
- If the NAME directive is omitted, the assembler will assume the primary name (first 256 characters) of the input source module file as the module name. The primary name is converted to capital letters for retrieval.
  If two or more module names are specified, the assembler will output a warning and ignore the second and subsequent module name declarations.
- A module name to be described in the operand field must not exceed 256characters.
- The uppercase and lowercase characters of a symbol name are distinguished.

[Example]

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


        CSEG
        MOV     A, B
        END
```

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

4.2.7 Branch instruction automatic selection directives

There are two unconditional branch instructions which write the branch address to the operand directly, "BR !addr20", and "BR $addr20".

With regard to these instructions, because the number of bytes for instructions differs, it is necessary for the user to use them after selecting which operand is suitable depending on the range of the branch destination in order to create a program with good memory efficiency.

Due to this, the 78K0R assembler has a directive to automatically select 2, 3 or 4-byte branch instructions depending on the range of the branch destination. This is called the branch destination instruction automatic selection directive.

The following branch instruction automatic selection directives are available.

Control Instruction	Overview
BR	Depending on the range of the value of the expression specified by the operand, the assembler automatically selects 2, 3 or 4-byte branch instructions and generates corresponding object code.
CALL	Depending on the range of the value of the expression specified by the operand, the assembler automatically selects 3 to 4-byte call branch instructions and generates corresponding object code.

BR

Tells the assembler to automatically select a 2-, 3-, or 4-byte BR branch instruction according to the value range of the expression specified in the operand field and to generate the object code applicable to the selected instruction.

[Description Format]

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

[Function]

- The BR directive tells the assembler to automatically select a 2-, 3-, or 4-byte BR branch instruction according to the value range of the expression specified in the operand field and to generate the object code applicable to the selected instruction.

[Use]

- Among the branch instructions listed below, the assembler determines the address range of the branch destination and automatically selects and outputs an instruction which uses the fewest number of bytes as much as possible. Use the BR directive if it is unclear whether a 2-byte branch instruction can be described.

Branch instruction	Explanation
“BR $addr20” (2 bytes)	Can be used if the address range of the branch destination is within the range of -80H to +7FH, from an address following the BR directive.
“BR !addr20” (3 bytes)	Can be used if the address range of the branch destination is within 64 KB.
“BR $!addr20” (3 bytes)	Calculates the displacement from the branch destination and can be used if the displacement is within the range of -8000H to +7FFFH
“BR !!addr20” (4 bytes)	Used in cases other than above

If an operand (branch destination) is located in a relocatable segment different from that to which the directive is located, and outside the BASE area, the directive will be substituted with a 4-byte instruction and the output.
If a directive and an operand (branch destination) are located in different segments and outside the BASE area, and their types are different, the directive will be substituted with a 4-byte instruction, even if the operand is located in an absolute segment.
If a directive and the branch destination are located in different segments and in the BASE area, the directive will be substituted with a 3-byte instruction (BR !addr20).

Remark The different type means the different relocatable segments if the BR directive is located in an absolute segment, or an absolute segment if the BR directive is located in a relocatable segment.

- If it is definitely known which of a2-, 3-, or 4-byte branch instruction should be described, describe the applicable instruction. This shortens the assembly time in comparison with describing the BR directive.

[Description]

- The BR directive can only be used within a code segment.
- The direct jump destination is described as the operand of the BR directive. "$" indicating the current location counter at the beginning of an expression cannot be described.
- For optimization, the following conditions must be satisfied.
  - No more than 1 label or forward-reference symbol in the expression.

- Do not describe an EQU symbol with the ADDRESS attribute.
- Do not describe an EQU defined symbol for "expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute".
- Do not describe an expression with ADDRESS attribute on which the HIGH/LOW/HIGHW/LOWW/ DATA-POS/BITPOS operator has been operated.

If these conditions are not met, the 4-byte BR instruction will be selected.

[Example]

```
ADDRESS                 NAME    SAMPLE
                C1      CSEG    AT      50H
00050H                  BR      L1                      ; (1)
00052H                  BR      L2                      ; (2)
00055H                  BR      L3                      ; (3)

0007DH          L1 :
0FFFFH          L2 :
10000H          L3 :

                C2      CSEG    AT      20050H
20050H                  BR      L4                      ; (4)
27FFFH          L4 :
                        END
```

(1) This BR directive generates a 2-byte branch instruction (BR $addr20) because the displacement between this line and the branch destination is within the range of -80H and +7FH.

(2) The branch destination of this BR directive is within 64 KB, so the BR directive will be substituted with a 3-byte branch instruction (BR !addr20).

(3) This BR directive will be substituted with the 4-byte branch instruction (BR !!addr20).

(4) This BR directive will be substituted with the 3-byte branch instruction (BR !addr20) because the displacement between this line and the branch destination is without the range of -8000H and +7FFFH.

CALL

Tells the assembler to automatically select a 3- or 4-byte CALL branch instruction according to the value range of the expression specified in the operand field and to generate the object code applicable to the selected instruction.

[Description Format]

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

[Function]

- The CALL directive tells the assembler to automatically select a 3- or 4-byte CALL branch instruction according to the value range of the expression specified in the operand field and to generate the object code applicable to the selected instruction.

[Use]

- Among the branch instructions listed below, the assembler determines the address range of the branch destination and automatically selects and outputs an instruction which uses the fewest number of bytes as much as possible. Use the CALL directive if it is unclear whether a 3-byte branch instruction can be described.

Branch Instruction	Explanation
“CALL !addr20“ (3 bytes)	Can be used if the address range of the branch destination is within 64 KB.
“CALL $!addr20“ (3 bytes)	Calculates the displacement from the branch destination and can be used if the displacement is within the range of -8000H to +7FFFH
“CALL !!addr20“ (4 bytes)	Used in cases other than above

If an operand (branch destination) is located in a relocatable segment different from that to which the directive is located, and outside the BASE area, the directive will be substituted with a 4-byte instruction and the output.
If a directive and an operand (branch destination) are located in different segments and outside the BASE area, and their types are different[Note], the directive will be substituted with a 4-byte instruction, even if the operand is located in an absolute segment.
If a directive and the branch destination are located in different segments and in the BASE area, the directive will be substituted with a 3-byte instruction (BR !addr20).

Note The different type means the different relocatable segments if the CALL directive is located in an absolute segment, or an absolute segment if the CALL directive is located in a relocatable segment.

- If it is definitely known which of a 3- or 4-byte branch instruction should be described, describe the applicable instruction. This shortens the assembly time in comparison with describing the CALL directive.

[Description]

- The CALL directive can only be used within a code segment.
- The direct jump destination is described as the operand of the CALL directive.
- For optimization, the following conditions must be satisfied.
  - No more than 1 label or forward-reference symbol in the expression.
  - Do not describe an EQU symbol with the ADDRESS attribute.
  - Do not describe an EQU defined symbol for "expression 1 with ADDRESS attribute - expression 2 with ADDRESS attribute".

- Do not describe an expression with ADDRESS attribute

If these conditions are not met, the 4-byte CALL instruction will be selected.

[Example]

```
ADDRESS                 NAME    SAMPLE
                C1      CSEG    AT      50H
00050H                  CALL    L1                      ; (1)
00053H                  CALL    L2                      ; (2)
08052H          L1 :
0FFFFH          L2 :


                C2      CSEG    AT      20050H
20050H                  CALL    L3                      ; (3)
27FFFH          L3 :
                        END
```

(1) The branch destination of this CALL directive is within 64 KB, so the CALL directive will be substituted with a 3-byte branch instruction (CALL !addr20).

(2) This CALL directive will be substituted with the 4-byte branch instruction (CALL !!addr20).

(3) This CALL directive will be substituted with the 3-byte branch instruction (CALL !addr20) because the displacement between this line and the branch destination is without the range of -8000H and +7FFFH.

4.2.8 Macro directives

When describing a source it is inefficient to have to describe for each series of high usage frequency instruction groups. This is also the source of increased errors.

Via macro directives, using macro functions it becomes unnecessary to describe many times to the same kind of instruction group series, and coding efficiency can be improved.

Macro basic functions are in substitution of a series of statements.

The following macro directives are available.

Control Instruction	Overview
MACRO	Executes a macro definition by assigning the macro name specified in the symbol field to a series of statements described between MACRO directive and the ENDM directive.
LOCAL	Declares that the symbol name specified in the operand column is a local symbol only effective in that macro body.
REPT	Only the value of the expression specified by the series of statements written between the REPT directive and the ENDM directive is developed repeatedly.
IRP	Only the number of actual arguments is repeatedly developed while the dummy argument is replaced by the actual argument specified by the operand in the series of statements between the IRP directive and ENDM directive.
EXITM	Develops the macro body defined with the MACRO directive, and also via REPT-ENDM, IRP-END M repeat is forced to complete.
ENDM	Completes a set of statements defined as a macro function.

MACRO

Executes a macro definition by assigning the macro name specified in the symbol field to a series of statements described between MACRO directive and the ENDM directive.

[Description Format]

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

[Function]

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

[Use]

- Define a series of frequently used statements in the source program with a macro name. After its definition only describe the defined macro name (see "(2) Referencing macros"), and the macro body corresponding to the macro name is expanded.

[Description]

- The MACRO directive must be paired with the ENDM directive.
- For the macro name to be described in the symbol field, see the conventions of symbol description in "(3) Symbol field".
- To reference a macro, describe the defined macro name in the mnemonic field.
- For the formal parameter(s) to be described in the operand field, the same rules as the conventions of symbol description will apply.
- Up to 16 formal parameters can be described per macro directive.
- Formal parameters are valid only within the macro body.
- An error occurs if any reserved word is described as a formal parameter. However, if a user-defined symbol is described, its recognition as a formal parameter will take precedence.
- The number of formal parameters must be the same as the number of actual parameters.
- A name or label defined within the macro body if declared with the LOCAL directive becomes effective with respect to one-time macro expansion.
- Nesting of macros (i.e., to refer to other macros within the macro body) is allowed up to eight levels including REPT and IRP directives.
- The number of macros that can be defined within a single source module is not specifically limited. In other words, macros may be defined as long as there is memory space available.
- Formal parameter definition lines, reference lines, and symbol names are not output to a cross-reference list.
- Two or more segments must not be defined in a macro body. If defined, an error will be output.

[Example]

```
        NAME    SAMPLE

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

        ADMAC   10H, 20H        ; (3)
        END
```

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

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

(3) Macro "ADMAC" is referenced.

LOCAL

Declares that the symbol name specified in the operand field is a local symbol that is valid only within the macro body

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- For the conventions on symbol names to be described in the operand field, see the conventions on symbol description in "(3) Symbol field".
- A symbol declared as LOCAL will be substituted with a symbol "??RAnnnn" (where n = 0000 to FFFF) at each macro expansion. The symbol "??RAnnnn" after the macro replacement will be handled in the same way as a global symbol and will be stored in the symbol table, and can thus be referenced under the symbol name "??RAnnnn".
- If a symbol is described within a macro body and the macro is referenced more than once, it means that the symbol would be defined more than once in the source module. For this reason, it is necessary to declare that the symbol is a local symbol that is valid only within the macro body.
- The LOCAL directive can be used only within a macro definition.
- The LOCAL directive must be described before using the symbol specified in the operand field (in other words, the LOCAL directive must be described at the beginning of the macro body).
- Symbol names to be defined with the LOCAL directive within a source module must be all different (in other words, the same name cannot be used for local symbols to be used in each macro).
- The number of local symbols that can be specified in the operand field is not limited as long as they are all within a line. However, the number of symbols within a macro body is limited to 64. If 65 or more local symbols are declared, the assembler will output an error and store the macro definition as an empty macro body. Nothing will be expanded even if the macro is called.
- Macros defined with the LOCAL directive cannot be nested.
- Symbols defined with the LOCAL directive cannot be called (referenced) from outside the macro.
- No reserved word can be described as a symbol name in the operand field. However, if a symbol same as the user-defined symbol is described, its recognition as a local symbol will take precedence.
- A symbol declared as the operand of the LOCAL directive will not be output to a cross-reference list and symbol table list.
- The statement line of the LOCAL directive will not be output at the time of the macro expansion.
- If a LOCAL declaration is made within a macro definition for which a symbol has the same name as a formal parameter of that macro definition, an error will be output.

[Example]

```
        NAME    SAMPLE

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

        ; Source text
REF1 :  MAC1                    ; (3)

??RA0000 :
        BR      $??RA0000       ; (2)

        BR      !LLAB           ; (4)     <- Error

REF2 :  MAC1                    ; (5)

??RA0001 :
        BR      $??RA0001       ; (2)
        END
```

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

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

(3) This macro reference calls macro MAC1.

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

(5) This macro reference calls macro MAC1.

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

```
        Assemble list

ALNO    STNO    ADRS    OBJECT  M I     SOURCE STATEMENT

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

REPT

Tells the assembler to repeatedly expand a series of statements described between this directive and the ENDM directive the number of times equivalent to the value of the expression specified in the operand field.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

[Example]

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

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

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

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

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

The REPT-ENDM block defined by statements (1) and (2) has been expanded three times.

On the assemble list, the definition statements (1) and (2) by the REPT directive in the source module is not displayed.

IRP

Tells the assembler to repeatedly expand a series of statements described between IRP directive and the ENDM directive the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters (from the left, the order) specified in the operand field.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

[Example]

```
NAME    SAMP1
CSEG

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

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

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

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

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

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

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

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

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

EXITM

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

[Description Format]

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

[Function]

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

[Use]

- This function is mainly used when a conditional assembly function (see "4.3.7 Conditional assembly control instructions") is used in the macro body defined with the MACRO directive.
- If conditional assembly functions are used in combination with other instructions in the macro body, part of the source program that must not be assembled is likely to be assembled unless control is returned from the macro by force using this EXITM directive. In such cases, be sure to use the EXITM directive.

[Description]

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

[Example]

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

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

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

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

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

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

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

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

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

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

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

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

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

Remark In the example here, conditional assembly control instructions are used. See "4.3.7 Conditional assembly control instructions". See "4.4 Macros" for the macro body and macro expansion.

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

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

The macro body of macro "MAC1" is expanded by referring to the macro in (11).

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

ENDM

Instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

[Example]

(1) MACRO-ENDM

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

(2) REPT-ENDM

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

(3) IRP-ENDM

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

4.2.9 Assemble termination directive

The assemble termination directive specifies completion of the source module to the assembler. This assembly termination directive must always be described at the end of each source module.

The assembler processes as a source module until the assemble completion directive. Consequently, with REPT block and IRP Block, if the assemble directive is before ENDM, the REPT block and IRP block become ineffective.

The following assemble termination directives are available.

Control Instruction	Overview
END	Declares termination of the source module

END

Declares termination of the source module

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

[Example]

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

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

4.3 Control Instructions

This chapter describes control instructions.

Control Instructions provide detailed instructions for assembler operation.

4.3.1 Overview

Control instructions provide detailed instructions for assembler operation and so are written in the source.

Control instructions do not become the target of object code generation.

Control instruction categories are displayed below.

Table 4-20. Control Instruction List

Control Instruction Type	Control Instruction
Assemble target type specification control instruction	PROCESSOR
Debug information output control instructions	DEBUG, NODEBUG, DEBUGA, NODEBUGA
Cross-reference list output specification control instructions	XREF, NOXREF, SYMLIST, NOSYMLIST
Include control instruction	INCLUDE
Assembly list control instructions	EJECT, LIST, NOLIST, GEN, NOGEN, COND, NOCOND, TITLE, SUBTITLE, FORMFEED, NOFORMFEED, WIDTH, LENGTH, TAB
Conditional assembly control instructions	IF, _IF, ELSEIF, _ELSEIF, ELSE, ENDIF, SET, RESET
Kanji code control instruction	KANJICODE
Other control instructions	TOL_INF, DGS, DGL

As with directives, control instructions are specified in the source.

Also, among the control instructions displayed in "Table 4-20. Control Instruction List", the following can be written as an assembler option even in the command line when the assembler is activated.

Table 4-21. Control Instructions and Assembler Options

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

4.3.2 Assemble target type specification control instruction

Assemble target type specification control instructions specify the assemble target type in the source module file. The following assemble target type specification control instructions are available.

Control Instruction	Overview
PROCESSOR	Specifies in a source module file the assemble target type.

PROCESSOR

Specifies in a source module file the assemble target type.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The PROCESSOR control instruction can be described only in the header section of a source module file. If the control instruction is described elsewhere, the assembler will be aborted.
- For the specifiable processor name, see the user's manual of the device used or "Device Files Operating Precautions".
- If the specified processor type differs from the actual target device subject to assembly, the assembler will be aborted.
- Only one PROCESSOR control instruction can be specified in the module header.
- The processor type of the target device subject to assembly may also be specified with the assembler option (-c) in the startup command line of the assembler. If the specified processor type differs between the source module file and the startup command line, the assembler will output a warning message and give precedence to the processor type specification in the startup command line.
- Even when the assembler option (-c) has been specified in the startup command line, the assembler performs a syntax check on the PROCESSOR control instruction.
- If the processor type is not specified in either the source module file or the startup command line, the assembler will be aborted.

[Application example]

```
$       PROCESSOR ( f1166a0 )
$       DEBUG
$       XREF

        NAME    TEST
          :
        CSEG
```

4.3.3 Debug information output control instructions

With debug information output control instructions is is possible to specify the output of debug information for the object module file in the source module file.

The following debug information output control instrucitons are available.

Control Instruction	Overview
DEBUG	Adds local symbol information in the object module file.
NODEBUG	Does not add local symbol information in the object module file.
DEBUGA	Adds assembler source debug information in the object module file.
NODEBUGA	Does not add assembler source debug information in the object module file.

DEBUG

Adds local symbol information in the object module file.

[Description Format]

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

[Function]

- The DEBUG control instruction tells the assembler to add local symbol information to an object module file.
- The NODEBUG control instruction tells the assembler not to add local symbol information to an object module file. However, in this case as well, the segment name is output to an object module file.

[Use]

- Use the DEBUG control instruction when symbolic debugging including local symbols is to be performed.

[Description]

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

NODEBUG

Does not add local symbol information in the object module file.

[Description Format]

```
[Δ]$[Δ]NODEBUG
[Δ]$[Δ]NODG           ; Abbreviated format
```

[Function]

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

[Use]

- Use the NODEBUG control instruction when:
- Symbolic debugging is to be performed for global symbols only
- Debugging is to be performed without symbols
- Only objects are required (as for evaluation with PROM)

[Description]

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

DEBUGA

Adds assembler source debug information in the object module file.

[Description Format]

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

[Function]

- The DEBUGA control instruction tells the assembler to add assembler source debugging information to an object module file.

[Use]

- Use the DEBUGA control instruction when debugging is to be performed at the assembler source level. An integrated debugger will be necessary for debugging at the source level.

[Description]

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

NODEBUGA

Does not add assembler source debug information in the object module file.

[Description Format]

```
[Δ]$[Δ]NODEBUGA
```

[Function]

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

[Use]

- Use the NODEBUGA control instruction when:
- Debugging is to be performed without the assembler source
- Only objects are required (as for evaluation with PROM)

[Description]

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

4.3.4 Cross-reference list output specification control instructions

cross-reference list output specification control instructions specify cross-reference list output in a source module file.

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

Control Instruction	Overview
XREF	Outputs a cross-reference list to an assemble list file.
NOXREF	Does not output a cross-reference list to an assemble list file.
SYMLIST	Outputs a symbol list to a list file.
NOSYMLIST	Does not output a symbol list to a list file.

XREF

Outputs a cross-reference list to an assemble list file.

[Description Format]

```
[Δ]$[Δ]XREF
[Δ]$[Δ]XR             ; Abbreviated format
```

[Function]

- The XREF control instruction tells the assembler to output a cross-reference list to an assembly list file.

[Use]

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

[Description]

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

NOXREF

Does not output a cross-reference list to an assemble list file.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

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

SYMLIST

Outputs a symbol list to a list file

[Description Format]

```
[Δ]$[Δ]SYMLIST
```

[Function]

- The SYMLIST control instruction tells the assembler to output a symbol list to a list file.

[Use]

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

[Description]

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

NOSYMLIST

Does not output a symbol list to a list file.

[Description Format]

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

[Function]

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

[Use]

- Use the NOSYMLIST control instruction not to output a symbol list.

[Description]

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

4.3.5 Include control instruction

Include control instructions are used when quoting other source module files in the source module
By using include control instructions effectively, the labor hours for describing source can be reduced.
The following include control instructions are available.

Control Instruction	Overview
INCLUDE	Quote a series of statements from another source module file.

INCLUDE

Quote a series of statements from another source module file.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The INCLUDE control instruction can only be described in ordinary source programs.
- The pathname or drive name of an INCLUDE file can be specified with the assembler option (-I).
- The assembler searches INCLUDE file read paths in the following sequence:

(1) When an INCLUDE file is specified without pathname specification

(a) Path in which the source file exists

(b) Path specified by the assembler option (-I)

(c) Path specified by the environment variable INC78K0R

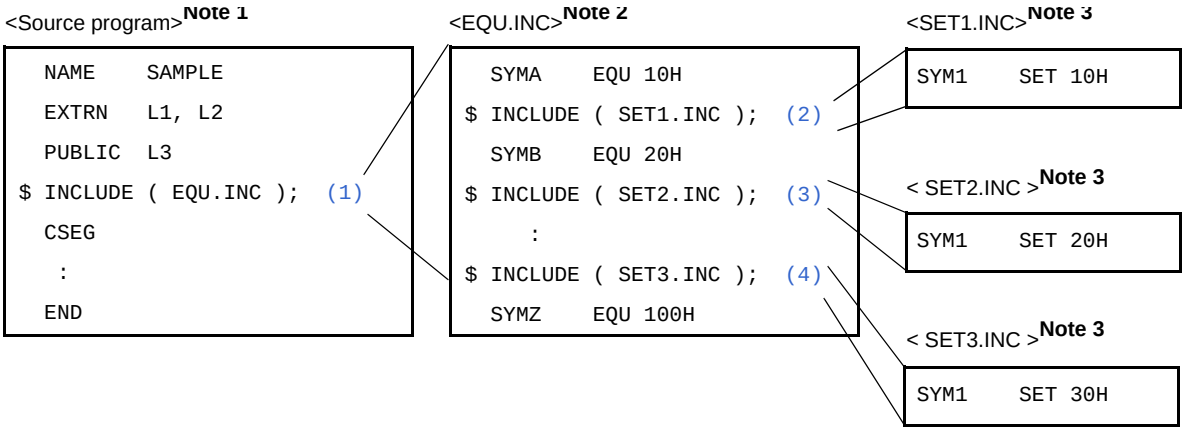
(2) When an INCLUDE file is specified with a pathname

If the INCLUDE file is specified with a drive name or a pathname which begins with backslash (\), the path specified with the INCLUDE file will be prefixed to the INCLUDE filename. If the INCLUDE file is specified with a relative path (which does not begin with \), a pathname will be prefixed to the INCLUDE filename in the order described in (1) above.

- Nesting of INCLUDE files is allowed up to seven levels. In other words, the nesting level display of INCLUDE files in the assembly list is up to 8 (the term "nesting" here refers to the specification of one or more other INCLUDE files in an INCLUDE file).
- The END directive need not be described in an INCLUDE file.
- If the specified INCLUDE file cannot be opened, the assembler will abort operation.
- An INCLUDE file must be closed with IF or _IF control instruction that is properly paired with an ENDIF control instruction within the INCLUDE file. If the IF level at the entry of the INCLUDE file expansion does not correspond with the IF level immediately after the INCLUDE file expansion, the assembler will output an error message and force the IF level to return to that level at the entry of the INCLUDE file expansion.

- When defining a macro in an INCLUDE file, the macro definition must be closed in the INCLUDE file. If an ENDM directive appears unexpectedly (without the corresponding MACRO directive) in the INCLUDE file, an error message will be output and the ENDM directive will be ignored. If an ENDM directive is missing for the MACRO directive described in the INCLUDE file, the assembler will output an error message but will process the macro definition by assuming that the corresponding ENDM directive has been described.
- Two or more segments cannot be defined in an include file. An error is output, if defined.

[Application example]



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

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

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

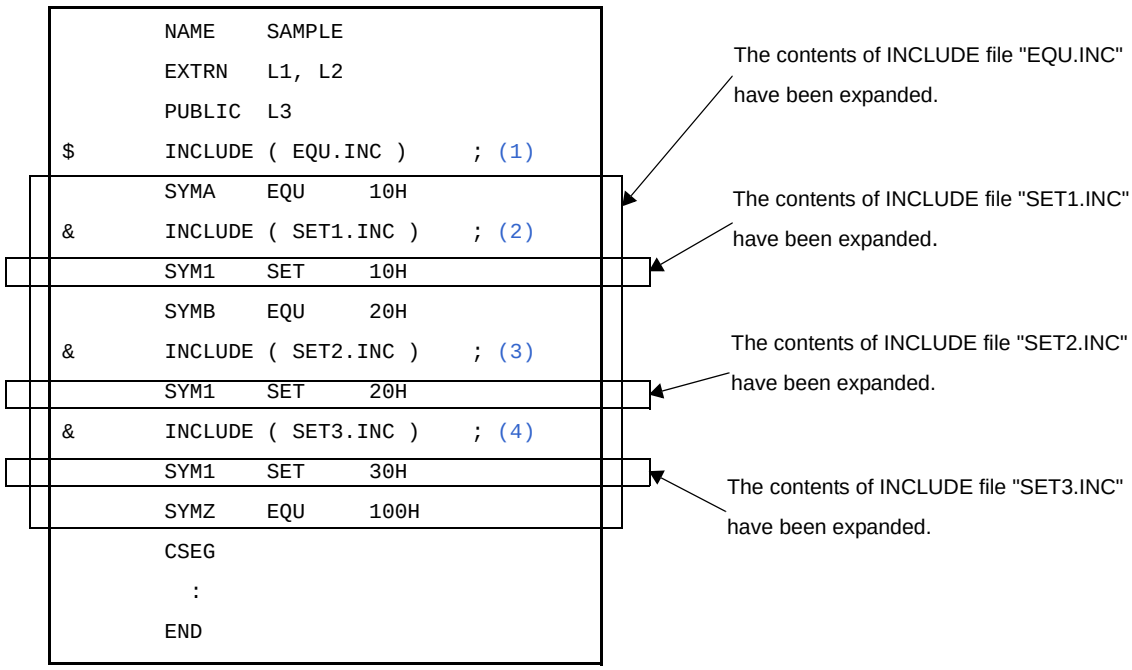
(4) This control instruction specifies "SET3.INC" as the INCLUDE file.

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

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

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

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



4.3.6 Assembly list control instructions

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

The following assembly list control instructions are available.

Control Instruction	Overview
EJECT	Indicates an Assembly List page break.
LIST	Indicates starting location of output of assembly list.
NOLIST	Indicates stop location of output of assembly list.
GEN	Outputs macro definition lines, reference line and also macro-expanded lines to assembly list.
NOGEN	Does not output macro definition lines, reference line and also macro-expanded lines to assembly list.
COND	Outputs approved and failed sections of the conditional assemble to the assembly list.
NOCOND	Does not output approved and failed sections of the conditional assemble to the assembly list
TITLE	Prints character strings in the TITLE column at each page header of an assembly list, symbol table list, or cross-reference list.
SUBTITLE	Prints character strings in the SUBTITLE column at header of an assembly list.
FORMFEED	Outputs form feed at the end of a list file.
NOFORMFEED	Does not output form feed at the end of a list file.
WIDTH	Specifies the maximum number of characters for one line of a list file.
LENGTH	Specifies the number of lines for 1 page of a list file.
TAB	Specifies the number of characters for list file tabs.

EJECT

Indicates an assembly list page break.

[Description Format]

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

[Default Assumption]

- EJECT control instruction is not specified.

[Function]

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

[Use]

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

[Description]

- The EJECT control instruction can only be described in ordinary source programs.
- Page ejection of the assembly list is executed after the image of the EJECT control instruction itself is output.
- If the assembler option (-np) or (-llo) is specified in the startup command line or if the assembly list output is disabled by another control instruction, the EJECT control instruction becomes invalid.
  See "78K0R Build" for those assembler options.
- If an illegal description follows the EJECT control instruction, the assembler will output an error message.

[Application example]

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

(1) Page ejection is executed with the EJECT control instruction.

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

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

LIST

Indicates starting location of output of assembly list.

[Description Format]

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

[Function]

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

[Use]

- LUse the LIST control instruction to cancel the assembly list output suppression specified by the NOLIST control instruction.
  By using a combination of NOLIST and LIST control instructions, you can control the amount of assembly list output as well as the contents of the list.

[Description]

- The LIST control instruction can only be described in ordinary source programs.
- If the LIST control instruction is specified after the NOLIST control instruction, statements described after the LIST control instruction will be output again on the assembly list. The image of the LIST or NOLIST control instruction will also be output on the assembly list.
- If neither the LIST nor NOLIST control instruction is specified, all statements in the source module will be output to an assembly list.

[Application example]

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

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

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

NOLIST

Indicates stop location of output of assembly list.

[Description Format]

```
[Δ]$[Δ]NOLIST
[Δ]$[Δ]NOLI           ; Abbreviated format
```

[Function]

- The NOLIST control instruction indicates to the assembler the line at which assembly list output must be suppressed.
  All source statements described after the NOLIST control instruction specification will be assembled, but will not be output on the assembly list until the LIST control instruction appears in the source program.

[Use]

- Use the NOLIST control instruction to limit the amount of assembly list output.
- Use the LIST control instruction to cancel the assembly list output suppression specified by the NOLIST control instruction.
  By using a combination of NOLIST and LIST control instructions, you can control the amount of assembly list output as well as the contents of the list.

[Description]

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

[Application example]

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

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

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

GEN

Outputs macro definition lines, reference line and also macro-expanded lines to assembly list.

[Description Format]

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

[Function]

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

[Use]

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

[Description]

- The GEN control instruction can only be described in ordinary source programs.
- If neither the GEN nor NOGEN control instruction is specified, macro definition lines, macro reference lines, and macro-expanded lines will be output to an assembly list.
- The specified list control takes place after the image of the GEN or NOGEN control instruction itself has been printed on the assembly list.
- If the GEN control instruction is specified after the NOGEN control instruction, the assembler will resume the output of macro-expanded lines.

[Application example]

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

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

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

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

NOGEN

Does not output macro definition lines, reference line and also macro-expanded lines to assembly list.

[Description Format]

```
[Δ]$[Δ]NOGEN
```

[Function]

- The NOGEN control instruction tells the assembler to output macro definition lines and macro reference lines but to suppress macro-expanded lines.

[Use]

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

[Description]

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

[Application example]

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

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

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

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

COND

Outputs approved and failed sections of the conditional assemble to the assembly list.

[Description Format]

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

[Function]

- The COND control instruction tells the assembler to output lines that have satisfied the conditional assembly condition as well as those which have not satisfied the conditional assembly condition to an assembly list.

[Use]

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

[Description]

- The COND control instruction can only be described in ordinary source programs.
- If neither the COND nor NOCOND control instruction is specified, the assembler will output lines that have satisfied the conditional assembly condition as well as those which have not satisfied the conditional assembly condition to an assembly list.
- The specified list control takes place after the image of the COND or NOCOND control instruction itself has been printed on the assembly list.
- If the COND control instruction is specified after the NOCOND control instruction, the assembler will resume the output of lines that have not satisfied the conditional assembly condition and lines in which IF/_IF, ELSEIF/_ELSEIF, ELSE, and ENDIF have been described.

[Application example]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )                  ; This part, though assembled, will not be outout
                                    ; to the list.
                MOV     A, #1H
$       ELSE                        ; This part, though assembled, will not be outout
                                    ; to the list.
                MOV     A, #0H      ; This part, though assembled, will not be outout
                                    ; to the list.
$       ENDIF                       ; This part, though assembled, will not be outout
                                    ; to the list.
        END
```

NOCOND

Does not output approved and failed sections of the conditional assemble to the assembly list.

[Description Format]

```
[Δ]$[Δ]NOCOND
```

[Function]

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

[Use]

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

[Description]

- The NOCOND control instruction can only be described in ordinary source programs.
- If neither the COND nor NOCOND control instruction is specified, the assembler will output lines that have satisfied the conditional assembly condition as well as those which have not satisfied the conditional assembly condition to an assembly list.
- The specified list control takes place after the image of the COND or NOCOND control instruction itself has been printed on the assembly list.
- The assembler increments the ALNO and STNO counts even after the list output control by the NOCOND control instruction.
- If the COND control instruction is specified after the NOCOND control instruction, the assembler will resume the output of lines that have not satisfied the conditional assembly condition and lines in which IF/_IF, ELSEIF/_ELSEIF, ELSE, and ENDIF have been described.

[Application example]

```
        NAME    SAMP

$       NOCOND
$       SET ( SW1 )
$       IF ( SW1 )                      ; This part, though assembled, will not be outout
                                        ; to the list.
                MOV     A, #1H
$       ELSE                            ; This part, though assembled, will not be outout
                                        ; to the list.
                MOV     A, #0H          ; This part, though assembled, will not be outout
                                        ; to the list.
$       ENDIF                           ; This part, though assembled, will not be outout
                                        ; to the list.
        END
```

TITLE

Prints character strings in the TITLE column at each page header of an assembly list, symbol table list, or cross-reference list.

[Description Format]

```
[Δ]$[Δ]TITLE[Δ]([Δ]'title-string'[Δ])
[Δ]$[Δ]TT[Δ]([Δ]'title-string'[Δ])        ; Abbreviated format
```

[Default Assumption]

- When the TITLE control instruction is not specified, the TITLE column of the assembly list header is left blank.

[Function]

- The TITLE control instruction specifies the character string to be printed in the TITLE column at each page header of an assembly list, symbol table list, or cross-reference list.

[Use]

- Use the TITLE control instruction to print a title on each page of a list so that the contents of the list can be easily identified.
- If you need to specify a title with the assembler option at each assembly time, you can save time and labor in starting the assembler by describing this control instruction in the source module file.

[Description]

- The TITLE control instruction can be described only in the header section of a source module file.
- If two or more TITLE control instructions are specified at the same time, the assembler will accept only the last specified TITLE control instruction as valid.
- Up to 60 characters can be specified as the title string. If the specified title string consists of 61 or more characters, the assembler will accept only the first 60 characters of the string as valid.
  However, if the character length specification per line of an assembly list file (a quantity "X") is 119 characters or less, "X - 60 characters" will be acceptable.
- If a single quotation mark (') is to be used as part of the title string, describe the single quotation mark twice in succession.
- If no title string is specified (the number of characters in the title string = 0), the assembler will leave the TITLE column blank.
- If any character not included in "(2) Character set" is found in the specified title string, the assembler will output "!" in place of the illegal character in the TITLE column.
- A title for an assembly list can also be specified with the assembler option (-lh) in the startup command line of the assembler.

[Application example]

```
$       PROCESSOR (f1166a0 )
$       TITLE ( 'THIS IS TITLE' )
        NAME    SAMPLE
        CSEG
        MOV     A, B
        END
```

The assemble list of the above application example is shown below. (with the number of lines per page specified as 72).

```
78K0R Assembler Vx.xx    THIS IS TITLE    Date:xx xxx xx  Page:1


Command :       -ll72 sample.asm
Para-file :
In-file :       sample.asm
Obj-file :      sample.rel
Prn-file :      sample.prn


         Assemble list


ALNO    STNO    ADRS    OBJECT  M I     SOURCE STATEMENT


 1       1                              $       PROCESSOR ( f1166a0 )
 2       2                              $       TITLE ( 'THIS IS TITLE' )
 3       3                                      NAME    SAMPLE
 4       4       ----                           CSEG
 5       5       00000   63                     MOV     A, B
 6       6                                      END


Segment information :


ADRS    LEN     NAME


00000   00001H  ?CSEG


Target chip : uPD78F1166_A0
Device file : Vx.xx
Assembly complete, 0 error(s) and 0 warning(s) found. (0)
```

SUBTITLE

Prints character strings in the SUBTITLE column at header of an assembly list.

[Description Format]

```
[Δ]$[Δ]SUBTITLE[Δ]([Δ]'title-string'[Δ])
[Δ]$[Δ]ST[Δ]([Δ]'title-string'[Δ])         ; Abbreviated format
```

[Default Assumption]

- When the SUBTITLE control instruction is not specified, the SUBTITLE section of the assembly list header is left blank.

[Function]

- The SUBTITLE control instruction specifies the character string to be printed in the SUBTITLE section at each page header of an assembly list.

[Use]

- Use the SUBTITLE control instruction to print a subtitle on each page of an assembly list so that the contents of the assembly list can be easily identified.
  The character string of a subtitle may be changed for each page.

[Description]

- The SUBTITLE control instruction can only be described in ordinary source programs.
- Up to 72 characters can be specified as the subtitle string.
  If the specified title string consists of 73 or more characters, the assembler will accept only the first 72 characters of the string as valid. A 2-byte character is counted as two characters, and tab is counted as one character.
- The character string specified with the SUBTITLE control instruction will be printed in the SUBTITLE section on the page after the page on which the SUBTITLE control instruction has been specified. However, if the control instruction is specified at the top (first line) of a page, the subtitle will be printed on that page.
- If the SUBTITLE control instruction has not been specified, the assembler will leave the SUBTITLE section blank.
- If a single quotation mark (') is to be used as part of the character string, describe the single quotation mark twice in succession.
- If the character string in the SUBTITLE section is 0, the SUBTITLE column will be left blank.
- If any character not included in "(2) Character set" is found in the specified subtitle string, the assembler will output "!" in place of the illegal character in the SUBTITLE column. If CR (0DH) is described, an error occurs and nothing will be output in the assembly list. If 00H is described, nothing from that point to the closing single quotation mark (') will be output.

[Application example]

```
        NAME    SAMP
        CSEG
$       SUBTITLE ( 'THIS IS SUBTITLE 1' )       ; (1)
$       EJECT                                   ; (2)
        CSEG
$       SUBTITLE ( 'THIS IS SUBTITLE 2' )       ; (3)
$       EJECT                                   ; (4)
$       SUBTITLE ( 'THIS IS SUBTITLE 3' )       ; (5)
        END
```

(1) This control instruction specifies the character string "THIS IS SUBTITLE 1".

(2) This control instruction specifies a page ejection.

(3) This control instruction specifies the character string "THIS IS SUBTITLE 2".

(4) This control instruction specifies a page ejection.

(5) This control instruction specifies the character string "THIS IS SUBTITLE 3".

The assembly list for this example appears as follows (with the number of lines per page specified as 80).

```
78K0R Assembler Vx.xx                           Date:xx xxx xx  Page:1

Command :       -cf1166a0 -ll80 sample.asm
Para-file :
In-file :       sample.asm
Obj-file :      sample.rel
Prn-file :      sample.prn
        Assemble list

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 1     1                        NAME SAMP
 2     2    -----                    CSEG
 3     3                     $   SUBTITLE ( 'THIS IS SUBTITLE 1' )   ; (1)
 4     4                     $   EJECT                               ; (2)
-------------------------------------------page ejection-------------------------
78K0R Assembler Vx.xx                           Date:xx xxx xx  Page:2

THIS IS SUBTITLE 1

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 5     5    -----                CSEG
 6     6                     $   SUBTITLE ( 'THIS IS SUBTITLE 2' )   ; (3)
 7     7                     $   EJECT                               ; (4)
```

```
-------------------------------------------page ejection-------------------------
78K0R Assembler Vx.xx                            Date:xx xxx xx  Page:3

THIS IS SUBTITLE 2

ALNO  STNO  ADRS  OBJECT  M I SOURCE STATEMENT

 8     8                    $   SUBTITLE ( 'THIS IS SUBTITLE 3' )   ; (5)
 9     9                        END

Segment informations :

ADRS  LEN     NAME

00000 00000H  ?CSEG

Target chip : uPD78F1166_A0
Device file : Vx.xx
Assembly complete, 0 error(s) and 0 warning(s) found. (0)
```

FORMFEED

Outputs form feed at the end of a list file.

[Description Format]

```
[Δ]$[Δ]FORMFEED
```

[Function]

- The FORMFEED control instruction tells the assembler to output a FORMFEED code at the end of an assembly list file.

[Use]

- Use the FORMFEED control instruction when you want to start a new page after printing the contents of an assembly list file.

[Description]

- The FORMFEED control instruction can be described only in the header section of a source module file.
- At the time of printing an assembly list, the last page of the list may not come out if printing ends in the middle of a page.
  In such a case, add a FORMFEED code to the end of the assembly list using the FORMFEED control instruction or assembler option (-lf).
  In many cases, a FORMFEED code will be output at the end of a file. For this reason, if a FORMFEED code exists at the end of a list file, an unwanted white page may be ejected. To prevent this, the NOFORMFEED control instruction or assembler option (-nlf) has been set by default value.
- If two or more FORMFEED/NOFORMFEED control instructions are specified at the same time, only the last specified control instruction will become valid.
- The output or non-output of a formfeed code may also be specified with the assembler option (-lf) or (-nlf) in the startup command line of the assembler.
- If the control instruction specification (FORMFEED/NOFORMFEED) in the source module differs from the specification (-lf/-nlf) in the startup command line, the specification in the startup command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the startup command line, the assembler performs a syntax check on the FORMFEED or NOFORMFEED control instruction.

NOFORMFEED

Does not output form feed at the end of a list file.

[Description Format]

```
[Δ]$[Δ]NOFORMFEED     ; Default assumption
```

[Function]

- The NOFORMFEED control instruction tells the assembler not to output a FORMFEED code at the end of an assembly list file.

[Use]

- Use the FORMFEED control instruction when you want to start a new page after printing the contents of an assembly list file.

[Description]

- The NOFORMFEED control instruction can be described only in the header section of a source module file.
- At the time of printing an assembly list, the last page of the list may not come out if printing ends in the middle of a page.
  In such a case, add a FORMFEED code to the end of the assembly list using the FORMFEED control instruction or assembler option (-lf).
  In many cases, a FORMFEED code will be output at the end of a file. For this reason, if a FORMFEED code exists at the end of a list file, an unwanted white page may be ejected. To prevent this, the NOFORMFEED control instruction or assembler option (-nlf) has been set by default value.
- If two or more FORMFEED/NOFORMFEED control instructions are specified at the same time, only the last specified control instruction will become valid.
- The output or non-output of a formfeed code may also be specified with the assembler option (-lf) or (-nlf) in the startup command line of the assembler.
- If the control instruction specification (FORMFEED/NOFORMFEED) in the source module differs from the specification (-lf/-nlf) in the startup command line, the specification in the startup command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the startup command line, the assembler performs a syntax check on the FORMFEED or NOFORMFEED control instruction.

WIDTH

Specifies the maximum number of characters for one line of a list file.

[Description Format]

```
[Δ]$[Δ]WIDTH[Δ]([Δ]columns-per-line[Δ])
```

[Default Assumption]

$WIDTH (132)

[Function]

- The WIDTH control instruction specifies the number of columns (characters) per line of a list file. "columns-per-line" must be a value in the range of 72 to 260.

[Use]

- Use the WIDTH control instruction when you want to change the number of columns per line of a list file.

[Description]

- The WIDTH control instruction can be described only in the header section of a source module file.
- If two or more WIDTH control instructions are specified at the same time, only the last specified control instruction will become valid.
- The number of columns per line of a list file may also be specified with the assembler option (-lw) in the startup command line of the assembler.
- If the control instruction specification (WIDTH) in the source module differs from the specification (-lw) in the startup command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the startup command line, the assembler performs a syntax check on the WIDTH control instruction.

LENGTH

Specifies the number of lines for 1 page of a list file

[Description Format]

```
[Δ]$[Δ]LENGTH[Δ]([Δ]lines-per-page[Δ])
```

[Default Assumption]

$LENGTH (66)

[Function]

- The LENGTH control instruction specifies the number of lines per page of a list file.
  "lines-per-page" may be "0" or a value in the range of 20 to 32767.

[Use]

- Use the LENGTH control instruction when you want to change the number of lines per page of a list file.

[Description]

- The LENGTH control instruction can be described only in the header section of a source module file.
- If two or more LENGTH control instructions are specified at the same time, only the last specified control instruction will become valid.
- The number of columns per line of a list file may also be specified with the assembler option (-ll) in the startup command line of the assembler.
- If the control instruction specification (LENGTH) in the source module differs from the specification (-ll) in the startup command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the startup command line, the assembler performs a syntax check on the LENGTH control instruction.

TAB

Specifies the number of characters for list file tabs.

[Description Format]

```
[Δ]$[Δ]TAB[Δ]([Δ]number-of-columns[Δ])
```

[Default Assumption]

$TAB (8)

[Function]

- The TAB control instruction specifies the number of columns as tab stops on a list file.
  "number-of-columns" may be a value in the range of 0 to 8.
- The TAB control instruction specifies the number of columns that becomes the basis of tabulation processing to output any list by replacing a HT (Horizontal Tabulation) code in a source module with several blank characters on the list.

[Use]

- Use HT code to reduce the number of blanks when the number of characters per line of any list is reduced using the TAB control instruction.

[Description]

- The TAB control instruction can be described only in the header section of a source module file.
- If two or more TAB control instructions are specified at the same time, only the last specified control instruction will become valid.
- The number of tab stops may also be specified with the assembler option (-lt) in the startup command line of the assembler.
- If the control instruction specification (TAB) in the source module differs from the specification (-lt) in the startup command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-np) has been specified in the startup command line, the assembler performs a syntax check on the TAB control instruction.

4.3.7 Conditional assembly control instructions

Conditonal assembly control instructions select, with the conditonal assemble switch, whether to to make a series of statements in the source module into an assemble target or not.

If conditional assemble instructions are made effective, it is possible to assemble without unnecessary statements and hardly changing the source module.

The following conditonal assembly control instructions are available.

Control Instruction	Overview
IF	Sets conditions in order to limit the assembly target source statements.
_IF	
ELSEIF	
_ELSEIF	
ELSE	
ENDIF	
SET	Sets value for switch name specified by IF/ELSEIF control instruction.
RESET	

IF

Sets conditions in order to limit the assembly target source statements.

[Description Format]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.
  Source statements described between the IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- If the evaluated value of the conditional expression or the switch name specified by the IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the IF condition is false (00H), source statements described after this IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/_ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- The ELSEIF or _ELSEIF control instruction is checked for true/false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).
  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- If the conditions of all the IF/_IF and ELSEIF/_ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.
- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression.
  Both IF/ELSEIF and _IF/_ELSEIF may be used in combination. In other words, ELSEIF/_ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).
  Up to five switch names can be used per module
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET/RESET control instruction.
  Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error occurs.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application example]

- Example 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ENDIF           ; (2)
          :
        END
```

(1) If the value of switch name "SW1" is true, statements in "text1" will be assembled.
If the value of switch name "SW1" is false, statements in "text1" will not be assembled.
The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".

(2) This instruction indicates the end of the source statement range for conditional assembly.

- Example 2

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF           ; (3)
          :
        END
```

(1) The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" is true, statements in "text1" will be assembled and statements in "text2" will not be assembled.

(2) If the value of switch name "SW1" in (1) is false, statements in "text1" will not be assembled and statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

- Example 3

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) The values of switch names "SW1", "SW2", and "SW3" have been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" or "SW2" is true, statements in "text1" will be assembled and statements in "text2", "text3", and "text4" will not be assembled.
If the values of switch names "SW1" and "SW2" are false, statements in "text1" will not be assembled and statements after (2) will be conditionally assembled.

(2) If the values of switch names "SW1" and "SW2" in (1) are false and the value of switch name "SW3" is true, statements in "text2" will be assembled and statements in "text1", "text3", and "text4" will not be assembled.

(3) If the values of switch names "SW1" and "SW2" in (1) and "SW3" in (2) are false and the value of switch statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be will not be assembled.

(4) If the values of switch names "SW1" and "SW2" in (1), "SW3" in (2), and "SW4" in (3) are all false, statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be assembled.

(5) This instruction indicates the end of the source statement range for conditional assembly.

_IF

Sets conditions in order to limit the assembly target source statements.

[Description Format]

```
[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.
  Source statements described between the _IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- If the evaluated value of the conditional expression or the switch name specified by the _IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the _IF condition is false (00H), source statements described after this _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/_ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- The ELSEIF or _ELSEIF control instruction is checked for true/false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).
  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- If the conditions of all the IF/_IF and ELSEIF/_ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.
- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression.
  Both IF/ELSEIF and _IF/_ELSEIF may be used in combination. In other words, ELSEIF/_ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).
  Up to five switch names can be used per module
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET/RESET control instruction.
  Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error occurs.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application example]

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
          :
        END
```

(1) The value of switch name "SYMA" has been defined with the EQU or SET directive described in "text0". If the symbol name "SYMA" is true (not "0"), statements in "text1" will be assembled and "text2" will not be assembled.

(2) If the value of the symbol name "SYMA" is "0", and "SYMB" and "SYMC" have the same value, statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

ELSEIF

Sets conditions in order to limit the assembly target source statements.

[Description Format]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.
  Source statements described between the IF or _IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- If the evaluated value of the conditional expression or the switch name specified by the IF or _IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the IF or _IF condition is false (00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/_ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- The ELSEIF or _ELSEIF control instruction is checked for true/false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).
  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- If the conditions of all the IF/_IF and ELSEIF/_ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.
- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression.
  Both IF/ELSEIF and _IF/_ELSEIF may be used in combination. In other words, ELSEIF/_ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).
  Up to five switch names can be used per module
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET/RESET control instruction.
  Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error occurs.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application example]

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) The values of switch names "SW1", "SW2", and "SW3" have been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" or "SW2" is true, statements in "text1" will be assembled and statements in "text2", "text3", and "text4" will not be assembled.
If the values of switch names "SW1" and "SW2" are false, statements in "text1" will not be assembled and statements after (2) will be conditionally assembled.

(2) If the values of switch names "SW1" and "SW2" in (1) are false and the value of switch name "SW3" is true, statements in "text2" will be assembled and statements in "text1", "text3", and "text4" will not be assembled.

(3) If the values of switch names "SW1" and "SW2" in (1) and "SW3" in (2) are false and the value of switch statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be will not be assembled.

(4) If the values of switch names "SW1" and "SW2" in (1), "SW3" in (2), and "SW4" in (3) are all false, statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be assembled.

(5) This instruction indicates the end of the source statement range for conditional assembly.

_ELSEIF

Sets conditions in order to limit the assembly target source statements.

[Description Format]

```
[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.
  Source statements described between the IF or _IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- If the evaluated value of the conditional expression or the switch name specified by the IF or _IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the IF or _IF condition is false (00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/_ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- The ELSEIF or _ELSEIF control instruction is checked for true/false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).
  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- If the conditions of all the IF/_IF and ELSEIF/_ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.
- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression.
  Both IF/ELSEIF and _IF/_ELSEIF may be used in combination. In other words, ELSEIF/_ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).
  Up to five switch names can be used per module
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET/RESET control instruction.
  Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error occurs.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application example]

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
          :
        END
```

(1) The value of switch name "SYMA" has been defined with the EQU or SET directive described in "text0". If the symbol name "SYMA" is true (not "0"), statements in "text1" will be assembled and "text2" will not be assembled.

(2) If the value of the symbol name "SYMA" is "0", and "SYMB" and "SYMC" have the same value, statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

ELSE

Sets conditions in order to limit the assembly target source statements.

[Description Format]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
or[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
or[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.
  Source statements described between the IF or _IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- If the evaluated value of the conditional expression or the switch name specified by the IF or _IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the IF or _IF condition is false (00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/_ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- The ELSEIF or _ELSEIF control instruction is checked for true/false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).
  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- If the conditions of all the IF/_IF and ELSEIF/_ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.
- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression.
  Both IF/ELSEIF and _IF/_ELSEIF may be used in combination. In other words, ELSEIF/_ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).
  Up to five switch names can be used per module
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET/RESET control instruction.
  Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error occurs.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application example]

- Example 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF           ; (3)
          :
        END
```

(1) The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" is true, statements in "text1" will be assembled and statements in "text2" will not be assembled.

(2) If the value of switch name "SW1" in (1) is false, statements in "text1" will not be assembled and statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

- Example 2

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) The values of switch names "SW1", "SW2", and "SW3" have been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" or "SW2" is true, statements in "text1" will be assembled and statements in "text2", "text3", and "text4" will not be assembled.
If the values of switch names "SW1" and "SW2" are false, statements in "text1" will not be assembled and statements after (2) will be conditionally assembled.

(2) If the values of switch names "SW1" and "SW2" in (1) are false and the value of switch name "SW3" is true, statements in "text2" will be assembled and statements in "text1", "text3", and "text4" will not be assembled.

(3) If the values of switch names "SW1" and "SW2" in (1) and "SW3" in (2) are false and the value of switch statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be will not be assembled.

(4) If the values of switch names "SW1" and "SW2" in (1), "SW3" in (2), and "SW4" in (3) are all false, statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be assembled.

(5) This instruction indicates the end of the source statement range for conditional assembly.

ENDIF

Sets conditions in order to limit the assembly target source statements.

[Description Format]

```
[Δ]$[Δ]IF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
or[Δ]$[Δ]_IFΔconditional-expression
        :
[Δ]$[Δ]ELSEIF[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
or[Δ]$[Δ]_ELSEIFΔconditional-expression
        :
[Δ]$[Δ]ELSE
        :
[Δ]$[Δ]ENDIF
```

[Function]

- The control instructions set the conditions to limit source statements subject to assembly.
  Source statements described between the IF or _IF control instruction and the ENDIF control instruction are subject to conditional assembly.
- If the evaluated value of the conditional expression or the switch name specified by the IF or _IF control instruction (i.e., IF or _IF condition) is true (other than 00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the IF or _IF condition is false (00H), source statements described after this IF or _IF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/_ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- The ELSEIF or _ELSEIF control instruction is checked for true/false status only when the conditions of all the conditional assembly control instructions described before this ELSEIF or _ELSEIF control instruction are not satisfied (i.e. the evaluated values are false).
  If the evaluated value of the conditional expression or the switch name specified by the ELSEIF or _ELSEIF control instruction (i.e. ELSEIF or _ELSEIF condition) is true, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will be assembled. For subsequent assembly processing, the assembler will proceed to the statement next to the ENDIF control instruction.
  If the ELSEIF or _ELSEIF condition is false, source statements described after this ELSEIF or _ELSEIF control instruction until the appearance of the next conditional assembly control instruction (ELSEIF/ _ELSEIF, ELSE, or ENDIF) in the source program will not be assembled.
- If the conditions of all the IF/_IF and ELSEIF/_ELSEIF control instructions described before the ELSE control instruction are not satisfied (i.e., all the switch names are false), source statements described after this ELSE control instruction until the appearance of the ENDIF control instruction in the source program will be assembled.
- The ENDIF control instruction indicates to the assembler the termination of source statements subject to conditional assembly.

[Use]

- With these conditional assembly control instructions, source statements subject to assembly can be changed without major modifications to the source program.
- If a statement for debugging that becomes necessary only during the program development is described in a source program, whether or not the debugging statement should be assembled (translated into machine language) can be specified by setting switches for conditional assembly.

[Description]

- The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression.
  Both IF/ELSEIF and _IF/_ELSEIF may be used in combination. In other words, ELSEIF/_ELSEIF may be used in a pair with IF or _IF and ENDIF.
- Describe absolute expression for a conditional expression.
- The rules of describing switch names are the same as the conventions of symbol description (for details, see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- If the two or more switch names are to be specified with the IF or ELSEIF control instruction, delimit each switch name with a colon (:).
  Up to five switch names can be used per module
- When two or more switch names have been specified with the IF or ELSEIF control instruction, the IF or ELSEIF condition is judged to be satisfied if one of the switch name values is true.
- The value of each switch name to be specified with the IF or ELSEIF control instruction must be defined with the SET/RESET control instruction.
  Therefore, if the value of the switch name specified with the IF or ELSEIF control instruction is not set in the source module with the SET or RESET control instruction in advance, it is assumed to be reset.
- If the specified switch name or conditional expression contains an illegal description, the assembler will output an error message and determine that the evaluated value is false.
- When describing the IF or _IF control instruction, the IF or _IF control instruction must always be paired with the ENDIF control instruction.
- If an IF-ENDIF block is described in a macro body and control is transferred back from the macro at that level by EXITM processing, the assembler will force the IF level to return to that level at the entry of the macro body. In this case, no error occurs.
- Description of an IF-ENDIF block in another IF-ENDIF block is referred to as nesting of IF control instructions. Nesting of IF control instructions is allowed up to 8 levels.
- In conditional assembly, object codes will not be generated for statements not assembled, but these statements will be output without change on the assembly list. If you do not wish to output these statements, use the $NOCOND control instruction.

[Application example]

- Example 1

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ENDIF           ; (2)
          :
        END
```

(1) If the value of switch name "SW1" is true, statements in "text1" will be assembled.
If the value of switch name "SW1" is false, statements in "text1" will not be assembled.
The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".

(2) This instruction indicates the end of the source statement range for conditional assembly.

- Example 2

```
        text0
$       IF ( SW1 )      ; (1)
                text1
$       ELSE            ; (2)
                text2
$       ENDIF           ; (3)
          :
        END
```

(1) The value of switch name "SW1" has been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" is true, statements in "text1" will be assembled and statements in "text2" will not be assembled.

(2) If the value of switch name "SW1" in (1) is false, statements in "text1" will not be assembled and statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

- Example 3

```
        text0
$       IF ( SW1 : SW2 )        ; (1)
                text1
$       ELSEIF ( SW3 )          ; (2)
                text2
$       ELSEIF ( SW4 )          ; (3)
                text3
$       ELSE                    ; (4)
                text4
$       ENDIF                   ; (5)
          :
        END
```

(1) The values of switch names "SW1", "SW2", and "SW3" have been set to true or false with the SET or RESET control instruction described in "text0".
If the value of switch name "SW1" or "SW2" is true, statements in "text1" will be assembled and statements in "text2", "text3", and "text4" will not be assembled.
If the values of switch names "SW1" and "SW2" are false, statements in "text1" will not be assembled and statements after (2) will be conditionally assembled.

(2) If the values of switch names "SW1" and "SW2" in (1) are false and the value of switch name "SW3" is true, statements in "text2" will be assembled and statements in "text1", "text3", and "text4" will not be assembled.

(3) If the values of switch names "SW1" and "SW2" in (1) and "SW3" in (2) are false and the value of switch statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be will not be assembled.

(4) If the values of switch names "SW1" and "SW2" in (1), "SW3" in (2), and "SW4" in (3) are all false, statements in "text4" will be assembled and statements in "text1", "text2", and "text3" will not be assembled.

(5) This instruction indicates the end of the source statement range for conditional assembly.

- Example 4

```
        text0
$       _IF ( SYMA )            ; (1)
                text1
$       _ELSEIF ( SYMB = SYMC ) ; (2)
                text2
$       ENDIF                   ; (3)
          :
        END
```

(1) The value of switch name "SYMA" has been defined with the EQU or SET directive described in "text0". If the symbol name "SYMA" is true (not "0"), statements in "text1" will be assembled and "text2" will not be assembled.

(2) If the value of the symbol name "SYMA" is "0", and "SYMB" and "SYMC" have the same value, statements in "text2" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly.

SET

Sets value for switch name specified by IF/ELSEIF control instruction.

[Description Format]

```
[Δ]$[Δ]SET[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
```

[Function]

- The SET control instructions give a value to each switch name to be specified with the IF or ELSEIF control instruction.
- The SET control instruction gives a true value (0FFH) to each switch name specified in its operand.

[Use]

- Describe the SET control instruction to give a true value (0FFH) to each switch name to be specified with the IF or ELSEIF control instruction.

[Description]

- With the SET and RESET control instructions, at least one switch name must be described.
  The conventions for describing switch names are the same as the conventions for describing symbols (see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- The specified switch name(s) may be the same as user-defined symbol(s) other than reserved words and other switch names.
- If two or more switch names are to be specified with the SET or RESET control instruction, delimit each switch name with a colon (:). Up to 1,000 switch names can be used per module.
- A switch name once set to "true" with the SET control instruction can be changed to "false" with the RESET control instruction, and vice versa.
- A switch name to be specified with the IF or ELSEIF control instruction must be defined at least once with the SET or RESET control instruction in the source module before describing the IF or ELSEIF control instruction.
- Switch names will not be output to a cross-reference list.

[Application example]

```
$       SET ( SW1 )             ; (1)
          :
$       IF ( SW1 )              ; (2)
                text1
$       ENDIF                   ; (3)
          :
$       RESET ( SW1 : SW2 )     ; (4)
          :
$       IF ( SW1 )              ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE                    ; (7)
                text4
$       ENDIF                   ; (8)
          :
        END
```

(1) This instruction gives a true value (0FFH) to switch name "SW1".

(2) Because the true value has been given to switch name "SW1" in (1) above, statements in "text1" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly that starts from (2).

(4) This instruction gives a false value (00H) to switch names "SW1" and "SW2", respectively.

(5) Because the false value has been given to switch name "SW1" in (4) above, statements in "text2" will not be assembled.

(6) Because the false value has also been given to switch name "SW2" in (4) above, statements in "text3" will not be assembled.

(7) Because both switch names "SW1" and "SW2" are false in (5) and (6) above, statements in "text4" will be assembled.

(8) This instruction indicates the end of the source statement range for conditional assembly that starts from (5).

RESET

Sets value for switch name specified by IF/ELSEIF control instruction.

[Description Format]

```
[Δ]$[Δ]RESET[Δ]([Δ]switch-name[[Δ]:[Δ]switch-name] ... [Δ])
```

[Function]

- The RESET control instructions give a value to each switch name to be specified with the IF or ELSEIF control instruction.
- The RESET control instruction gives a false value (00H) to each switch name specified in its operand.

[Use]

- Describe the RESET control instruction to give a false value (00H) to each switch name to be specified with the IF or ELSEIF control instruction.

[Description]

- With the SET and RESET control instructions, at least one switch name must be described.
  The conventions for describing switch names are the same as the conventions for describing symbols (see "(3) Symbol field").
  However, the maximum number of characters that can be recognized as a switch name is always 31.
- The specified switch name(s) may be the same as user-defined symbol(s) other than reserved words and other switch names.
- If two or more switch names are to be specified with the SET or RESET control instruction, delimit each switch name with a colon (:). Up to 1,000 switch names can be used per module.
- A switch name once set to "true" with the SET control instruction can be changed to "false" with the RESET control instruction, and vice versa.
- A switch name to be specified with the IF or ELSEIF control instruction must be defined at least once with the SET or RESET control instruction in the source module before describing the IF or ELSEIF control instruction.
- Switch names will not be output to a cross-reference list.

[Application example]

```
$       SET ( SW1 )             ; (1)
          :
$       IF ( SW1 )              ; (2)
                text1
$       ENDIF                   ; (3)
          :
$       RESET ( SW1 : SW2 )     ; (4)
          :
$       IF ( SW1 )              ; (5)
                text2
$       ELSEIF ( SW2 )          ; (6)
                text3
$       ELSE                    ; (7)
                text4
$       ENDIF                   ; (8)
          :
        END
```

(1) This instruction gives a true value (0FFH) to switch name "SW1".

(2) Because the true value has been given to switch name "SW1" in (1) above, statements in "text1" will be assembled.

(3) This instruction indicates the end of the source statement range for conditional assembly that starts from (2).

(4) This instruction gives a false value (00H) to switch names "SW1" and "SW2", respectively.

(5) Because the false value has been given to switch name "SW1" in (4) above, statements in "text2" will not be assembled.

(6) Because the false value has also been given to switch name "SW2" in (4) above, statements in "text3" will not be assembled.

(7) Because both switch names "SW1" and "SW2" are false in (5) and (6) above, statements in "text4" will be assembled.

(8) This instruction indicates the end of the source statement range for conditional assembly that starts from (5).

4.3.8 Kanji code control instruction

This is a control instruction which specifies which character code to use to interpret kanji characters described in the comment.

The following kanji code control instructions are available.

Control Instruction	Overview
KANJICODE	Interprets Kanji character code for specified Kanji characters described in the comment.

KANJICODE

Interprets Kanji character code for specified Kanji characters described in the comment.

[Description Format]

```
[∆]$[∆]KANJICODE[∆]kanji-code
```

[Default Assumption]

$KANJICODE SJIS

[Function]

- Interprets Kanji character code for specified Kanji characters described in the comment.
- A kanji cord can describe SJIS/EUC/NONE.

  SJIS : Interpreted as a Shift JIS code.
  EUC : Interpreted as a EUC code.
  NONE : Not interpreted as a kanji.

[Use]

- Use to specify the interpretation of the kanji code (2-byte code) of the kanji (2-byte character) in the comment line.

[Description]

- The KANJICODE control instruction can be described only in the header section of a source module file.
- If two or more KANJICODE control instructions are specified in the header section of a source module file at the same time, only the last specified control instruction will become valid.
- Kanji code specification stops may also be specified with the assembler option (-zs/-ze/-zn) in the startup command line of the assembler.
- If the control instruction specification (KANJICODE) in the source module differs from the specification (-zs/-ze/-zn) in the startup command line, the specification in the command line will take precedence over that in the source module.
- Even when the assembler option (-zs/-ze/-zn) has been specified in the startup command line, the assembler performs a syntax check on the KANJICODE control instruction.

4.3.9 Other control instructions

The control instructions shown below are special control instructions output by an upper level program such as C compiler.

- $TOL_INF
- $DGS
- $DGL

4.4 Macros

This section explains how to use macros.

Macros are very useful when you need to use a series of statements repeatedly in a source program.

4.4.1 Overview

Macros make it easy to repeat a series of statements over and over again in a source program.

A macro contains a series of instructions in a macro body, which is defined between MACRO and ENDM directives. These instructions are inserted into the source program at any location that references the macro.

Macros make it easier to write source programs. They are different from subroutines.

The difference between macros and subroutines is explained below. For effective use, select either a macro or a subroutine according to the specific purpose.

(1) Subroutines

- Subroutines handle processing that must be repeated many times in a program. A subroutine is converted into machine language once by the assembler.
- To use a subroutine, simply call it with a subroutine call instruction. (Usually you will also need to set the arguments of the subroutine before calling it, and adjust for them afterwards.)
  Effective use of subroutines enables program memory to be used with high efficiency.
- Subroutines are also important in structured programming. Dividing the program into functional blocks clarifies the overall structure of the program and makes it easier to understand. There are benefits for design, coding, and maintenance.

(2) Macros

- The basic function of macros is to replace a series of instructions with a macro name.
  A macros is defined as a series of instruction between MACRO and ENDM directives. When the macro name appears in the source code, the instructions are inserted at that location. The assembler replaces any formal parameters of the macro with actual parameters and converts the instructions into machine language.
- Macros can have parameters.
  For example, if there are instruction groups that are the same in processing procedure but are different in the data to be described in the operand, define a macro by assigning formal parameter(s) to the data. By describing the macro name and the actual parameter(s) at macro reference time, the assembler can cope with various instruction groups that differ only in part of the statement description.

Subroutines are used mainly to reduce memory requirements and clarify the structure of programs. Macros are used to make programs easier to describe and understand.

4.4.2 Using macros

(1) Macro definitions

Use the MACRO and ENDM directives to define macros.

(a) Format

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

(b) Function

Define a macro, assigning the macro name specified in the symbol field to the series of statements (called the macro body) between the MACRO directive and the ENDM directive

(c) Example

```
ADMAC   MACRO   PARA1, PARA2
        MOV     A, #PARA1
        ADD     A, #PARA2
        ENDM
```

The above example shows a simple example that adds the two values PARA1 and PARA2, and stores the result in register A. The macro is named ADMAC. PARA1 and PARA2 are formal parameters.
For details, see "4.2.8 Macro directives".

(2) Referencing macros

To reference an already defined macro, specify the macro name in the mnemonic field.

(a) Format

Symbol field	Mnemonic field	Operand field	Comment field
[*label*:]	*macro-name*	[*actual-parameter*[, ...]]	[; comment]

(b) Function

Reference the macro body to which the specified name has been assigned.

(c) Use

Use the above format to reference a macro body.

(d) Explanation

- The macro name specified in the mnemonic field must have been defined before the macro reference.
- Delimit actual parameters with commas (,). Up to 16 actual parameters can be specified, provided that they all appear in the same line.
- Space characters cannot appear in an actual parameter string.
- If an actual parameter string contains a comma (,), semicolon (;), blank, or a tab, enclose the string in single quotation marks (').
- Formal parameter are converted to actual parameters from the left, in the order that they occur in the macro definition. The number of actual parameters must match the number of formal parameters. If it does not, a warning is issued.

(e) Example

```
        NAME    SAMPLE
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM

        CSEG
          :
        ADMAC   10H, 20H
          :
        END
```

The already defined macro "ADMAC" is referenced.
10H and 20H are actual parameters.

(3) Macro expansion

The assembler processes a macro as follows:

- When it encounters a macro name, the assembler inserts the corresponding macro body at the location of the macro name.
- The inserted macro body is then assembled in the same way as other stateme

(4) Example

When the macro shown above in "(2) Referencing macros" is referenced, it is expanded as shown below.

```
        NAME    SAMPLE

        ; Macro definition
ADMAC   MACRO   PARA1, PARA2
                MOV     A, #PARA1
                ADD     A, #PARA2
        ENDM

        ; Source code
        CSEG
          :

        ; Macro expansion
        ADMAC   10H, 20H              ; (a)
        MOV     A, #10H
        ADD     A, #20H
        ; Source code
          :
        END
```

(a) The macro body is inserted when the macro name is referenced.
In the macro body, formal parameters are replaced by actual parameters.

4.4.3 Symbols in macros

Two types of symbols can be defined in macros: global symbols and local symbols.

(1) Global symbols

- A global symbol is a symbol that can be referenced from any statement in a source program. Therefore, if a macro that defines a global symbol is referenced more than once when expanding a series of statements, a double definition error will occur.
- Symbols not defined with the LOCAL directive are global symbols.

(2) Local symbols

- A local symbol is a symbol defined with the LOCAL directive (see "4.2.8 Macro directives").
- A local symbol can be referenced only within the macro that declares it as LOCAL.
- No local symbol can be referenced from outside the macro that declares it

<Application example>

```
        NAME    SAMPLE
          ; Macro definition
MAC1    MACRO
        LOCAL   LLAB            ; (a)
LLAB :                          ; (b)
          :
GLAB :                          ; (c)
        BR      LLAB            ; (d)
        BR      GLAB            ; (e)
        ENDM
          :
          ; Source code
REF1 :  MAC1                    ; (f) <- Macro reference
          :
        BR      LLAB            ; (g) <- Error

REF2 :  MAC1                    ; (h) <- Macro reference
          :
GLAB :                          ; (i) <- Error
          :
        END
```

(a) This declares label LLAB as a local symbol.

(b) This defines label LLAB as a local symbol.

(c) This defines label GLAB as a global symbol.

(d) This references the local symbol LLAB in macro MAC1.

(e) This references the global symbol GLAB from inside the definition of macro MAC1.

(f) This references macro MAC1.

(g) This reference the local symbol LLAB from outside the definition of macro MAC1, causing an error when the program is assembled.

(h) This references macro MAC1
The same macro is referenced twice.

(i) This defines the label GLAB as a global symbol.
The same label is already defined, so this causes an error when the program is assembled.

<Assemble list for the above application example>

```
        NAME    SAMPLE
          :
REF1 :  MAC1
          ; Macro expansion
??RA0000 :
          :
GLAB :                          <- Error
        BR      ??RA0000
        BR      GLAB
          ; Source code
          :
        BR      !LLAB           <- Error
        BR      !GLAB
          :
REF2 :  MAC1
          ; Macro expansion
??RA0001 :
          :
GLAB :                          <- Error
        BR      ??RA0001
        BR      GLAB
          ; Source code
          :
        END
```

Macro MAC1 defines the global symbol GLAB.

Macro MAC1 is referenced twice. An error occurs when the macro is expanded the second time, because the global symbol GLAB is defined twice.

4.4.4 Macro operators

There are two macro operators: "&" (ampersand) and ", "' (single quotation mark).

(1) & (Concatenation)

- The ampersand "&" concatenates one character string to another within a macro body.
  At macro expansion time, the character string on the left of the ampersand is concatenated to the character string on the right of the sign. The "&" itself disappears after concatenating the strings.
- At macro definition time, strings before and after "&" in symbols are recognized as local symbols and formal parameters. At macro expansion time, strings before and after the "&" in the symbol are evaluated as the local symbols and formal parameters, and concatenated into single symbols.
- The "&" sign enclosed in a pair of single quotation marks is simply handled as data.
- Two "&" signs in succession are handled as a single "&" sign.

Following is an application example.

(a) Macro definition

```
MAC     MACRO   P
LAB&P :                 <- Formal parameter P is recognized
        D&B     10H
        DB      'P'
        DB      P
        DB      '&P'
        ENDM
```

(b) Macro reference

```
        MAC     1H
LAB1H :
        DB      10H     <- D&B has been concatenated to DB
        DB      'P'
        DB      1H
        DB      '&P'    <- Quoted '&' is simply data
```

(2) ' (Single quotation mark)

- If a character string enclosed by a pair of single quotation marks appears at the beginning of an actual parameter in a macro reference line or an IRP directive, or if it appears after a delimiting character, the character string is interpreted as an actual parameter. The character string is passed as a actual parameter without the enclosing single quotation marks.
- If a character string enclosed by a pair of single quotation marks appears in a macro body, it is simply handled as data.
- To use a single quotation mark as a single quotation mark in text, write the single quotation mark twice in succession.

<Application example>

```
        NAME    SAMP
MAC1    MACRO   P
        IRP     Q, <P>
                MOV     A, #Q
        ENDM
ENDM


        MAC1    '10, 20, 30'
```

When the source code in the above example is assembled, macro MAC1 is expanded as shown below.

```
IRP     Q, <10, 20, 30>
        MOV A, #Q
ENDM
        MOV     A, #10          ; IRP expansion
        MOV     A, #20          ; IRP expansion
        MOV     A, #30          ; IRP expansion
```

4.5 Reserved Words

Reserved words are character strings reserved in advance by the assembler. They include machine language instructions, directives, control instructions, operators, register names, and sfr symbols. Reserved words cannot be used for other than the intended purposes.

The following tables explain where reserved words can appear in source code statements and list the words reserved by the assembler.

Table 4-22. Fields Where Reserved Words Can Appear

Field	Explanation
Symbol field	No reserved words can appear in this field.
Mnemonic field	Only machine language instructions and directives can appear in this field.
Operand field	Only operators, sfr symbols, and register names can appear in this field.
Comment field	All reserved words can appear in this field.

Table 4-23. List of Reserved Words

Type	Reserved Word
Operators	AND, BITPOS, DATAPOS, EQ (=), GE (>=), GT (>), HIGH, HIGHW, LE (<=), LOW, LOWW, LT (<), MASK, MOD, NE (< >), NOT, OR, SHL, SHR, XOR
Directives	AT, BASE, BASEP, BR, BSEG, CALL, CALLT0, CSEG, DB, DBIT, DG, DS, DSEG, DSPRAM, DW, END, ENDM, ENDS, EQU, EXITM, EXTBIT, EXTRN, FIXED, IHRAM, IRP, IXRAM, LOCAL, LRAM, MACRO, MIRRORP, NAME, OPT_BYTE, ORG, PAGE64KP, PUBLIC, REPT, SADDR, SADDRP, SECUR_ID, SET, UNIT, UNIT64KP, UNITP
Control instructions	COND, NOCOND, DEBUG, NODEBUG, DEBUGA [DG], NODEBUGA [NODG], EJECT [EJ], FORMFEED, NOFORMFEED, GEN, NOGEN, IF, _IF, ELSEIF, _ELSEIF, ELSE, ENDIF, INCLUDE [IC], KANJICODE, LENGTH, LIST [LI], NOLIST [NOLI], PROCESSOR [PC], SET, RESET, SUBTITLE [ST], SYMLIST, NOSYMLIST, TAB, TITLE [TT], WIDTH, XREF[XR], NOXREF [NOXR]
Other	DGL, DGS, SFR, SFRP, TOL_INF

Remark Items in brackets following control instructions indicate the abbreviated format.

See the user's manual of the target device for a list of sfr names, a list of interrupt request sources, and lists of machine language instructions and register names.

4.6 Instructions

This section explains the functions of 78K0R microcontroller instructions.

4.6.1 Differences from 78K0 microcontrollers (for assembler users)

- The new pipeline architecture reduces the number of clock cycles for all instructions. Existing programs must be re-evaluated.
- All instruction code maps have changed. Programs must be reassembled. When you reassemble, the code size is likely to increase, but in some cases the overall code size may shrink if old instructions are replaced with new ones.
- The memory space has changed from 64 KB to 1 MB, allowing greater stack depth. Address changes are required if assembly programs manipulate RAM that is accessed with the stack pointer. Depending on the multiple CALL and multiple interrupt depth, the stack size should be set to slightly more than is normally required.
- The CALLT table has been moved from [0040H to 007FH] to [0080H to 00BFH]. References to CALLT table addresses must be changed.
- Assembler programs must be rewritten if they utilized the bank switching mechanisms of 78K0 microcontrollers.
- Addresses have changed when expansion RAM is used. Update these addresses.
- If your programs execute instructions from expansion RAM, they will be affected by changes in memory space addresses. Change BR !addr16 statements to BR !!addr20, and change CALL !addr16 statements to CALL !!addr20.
- There are no IMS or IXS registers (used to set memory space). Unless external memory is used, code that uses those registers must be deleted. If external memory is used, note that the specifications of the MM/MEM registers (used to set memory space) have changed. For details, refer to the user's manual of the target microcontroller.
- The following instructions have been deleted, and alternative code is output, resulting in code size increases. These instructions can still be used by specifying the -compati option, but the assembler replaces them automatically with the replacement code.

Instruction	Operand	Remarks
DIVUW	C	The alternative instructions divide while shifting, which increases execution time. A shift instruction has been added, so it is recommended that this instruction be changed to the new shift instruction.
ROR4	[HL]	The alternative instructions take longer to execute. A shift instruction has been added, so it is recommended that this instruction be changed to the new shift instruction.
ROL4	[HL]	The alternative instructions take longer to execute. A shift instruction has been added, so it is recommended that this instruction be changed to the new shift instruction.
ADJBA	None	The alternative instructions take longer to execute. There is no equivalent additional instruction.
ADJBS	None	The alternative instructions take longer to execute. There is no equivalent additional instruction.
CALLF	!addr11	CALLF is changed automatically to a 3-byte CALL !addr16 instruction. This instruction can still be used with no modifications required.
DBNZ	B, $addr16 C, $addr16 saddr, $addr16	This instruction is split into the following instructions: DEC B / DEC C / DEC saddr, and BNZ $addr20. This instruction can still be used with no modifications required.

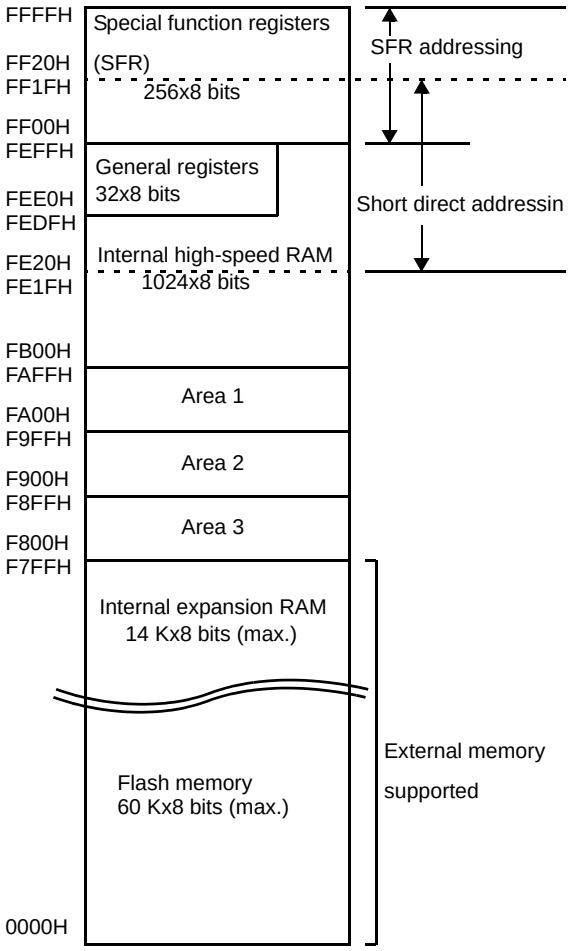
4.6.2 Memory space

(1) Memory space

The memory space of 78K0 microcontrollers was limited to 64 KB, but this has been expanded to 1 MB in 78K0 microcontrollers.

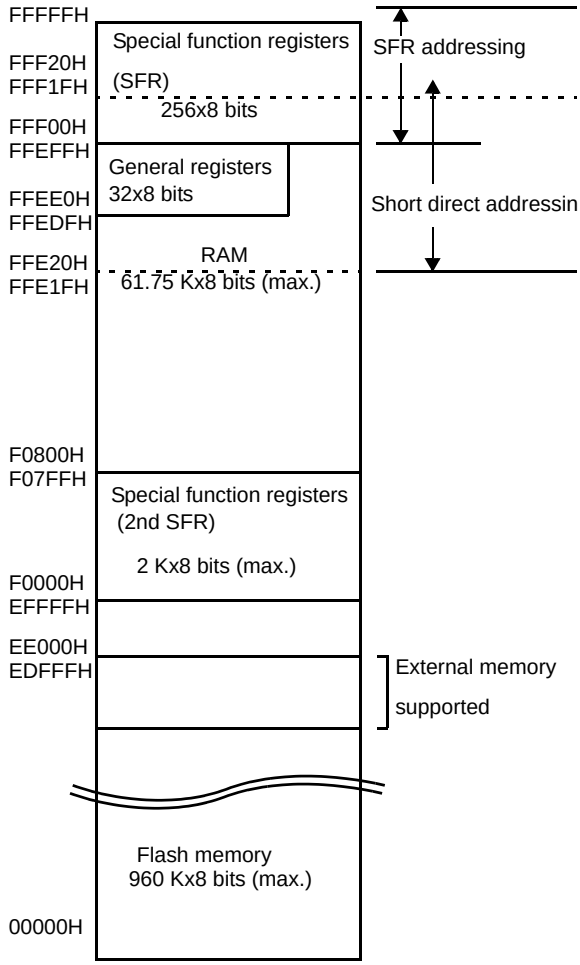
Figure 4-8. Memory Maps of 78K0 and 78K0R Microcontrollers

< 78K0 microcontrollers >



- Program memory space is 60 KB (max.).
- Internal high-speed RAM area is 1 KB (max.) (stacken-abled).
  Internal expansion RAM area is 14 KB (max.) (fetchen-abled).
- Area 1, area 2, and area 3 are from F800H to FAFFH (fixed).
- External expansion memory supported.

< 78K0R microcontrollers >



- Program memory space is 960 KB (max.).
- RAM space is 61.75 KB (max.) (stack enabled, fetchen-abled).
- 2nd SFR area (name changed) is 2 KB (max.),from F0000H to F07FFH.
- Supports external expansion memory.
  The external expansion memory space can be allocated from the product-mounted flash memory area to EDFFFH.

(2) Internal program memory space

In 78K0R microcontrollers, the address range of program memory space is from 00000H to EFFFFH.

For information about the maximum size of internal ROM (flash memory), refer to the user's manual of the target microcontroller.

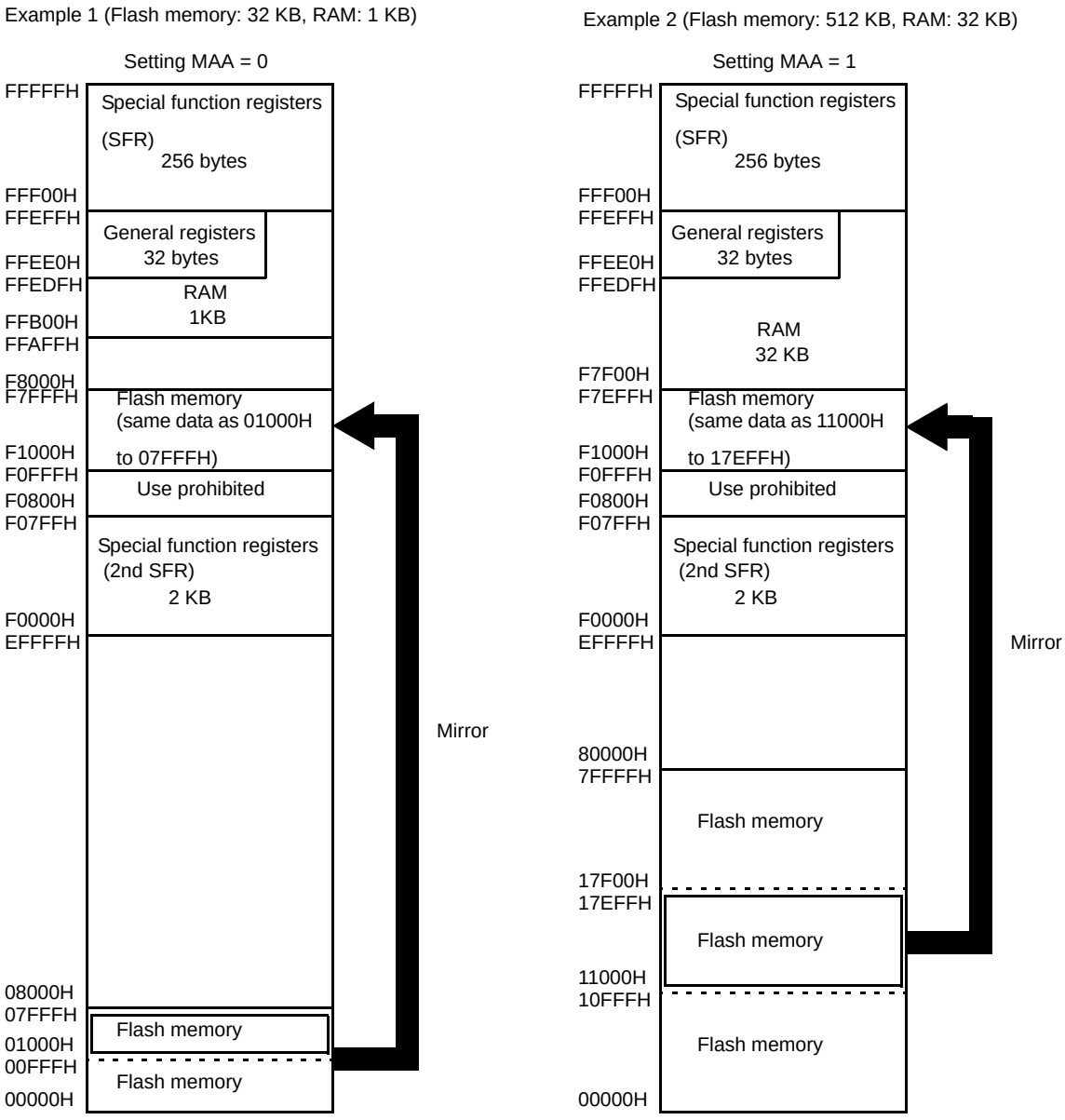
(a) Mirror area

In the 78K0R microcontrollers, the data flash areas from 00000H to 0FFFFH (when MAA = 0) and from 10000H to 1FFFFH (when MAA = 1) are mirrored to the addresses from F0000H to FFFFFH. Data flash contents can be read with shorter code by reading from F0000H to FFFFFH. However, data flash areas are not mirrored to the SFR, second SFR, RAM, and use-prohibited areas.

Mirror areas can only be read, and instruction fetch is not possible.

The following figure shows two examples. Specifications vary for each product, so for details refer to the user's manual of the target microcontroller.

Figure 4-9. Mirror Area Examples



Remark MAA: Bit 0 of the processor mode control register (PMC). (For details, see "(a) Processor mode control register (PMC)".)

(b) Vector table area

In the 78K0R microcontrollers, the 128-byte area from 0000H to 007FH is reserved as the vector table area. The number of interrupts is 61 (maximum) + RESET vector + on-chip debugging vector + software break vector. Since a vector is only 2 bytes of code, the interrupt branch destination address range is 64 KB from 00000H to 0FFFFH. In the 78K0 microcontrollers, addresses from 0040H to 007FH are used for the CALLT table, but in the 78K0R microcontrollers they have been changed to vector addresses.

(c) CALLT instruction table area

In the 78K0R microcontrollers, the 64-byte area from 0080H to 00BFH is reserved as the CALLT instruction table area.

In 78K0 microcontrollers, the CALL instruction is 1 byte, but 78K0R microcontrollers have 2-byte CALL instructions. Addresses have also changed.

Since address codes are only 2 bytes long, interrupt branch destination addresses are the 64 KB from 00000H to 0FFFFH.

(3) Internal data memory (internal RAM) space

78K0 microcontrollers have internal high-speed RAM and internal expansion RAM. The internal high-speed RAM can be used for stack, and the internal expansion RAM can be used for fetch. By contrast, 78K0R microcontrollers have just one RAM area that can be used for both stack and fetch.

The upper address limit is fixed at FFEFFH, and the lower limit extends downward according to the amount of RAM mounted on the product. The maximum size is 61.75 KB. For more information about the lower limit, refer to the user's manual of the target microcontroller.

The general register area and saddr space (from FFEE0H to FFEFFH) have the same addresses in the 78K0 series and the 78K0R series. The stack can be located anywhere within the mounted RAM.

(4) Special function register (SFR) area

Unlike general registers, SFRs have special functions.

The SFR space is allocated to the area from FFF00H to FFFFFH.

Although SFR specifications are the same as for 78K0 microcontrollers, there have been changes that affect some registers that had fixed addresses in the 78K0 series. For details, refer to the user's manual of the target microcontroller.

(5) Extended SFR (2nd SFR) area

Unlike general registers, 2nd SFRs have special functions.

The , 2nd SFR space is from F0000H to F07FFH. SFRs outside of the SFR area (from FFF00H to FFFFFH) are assigned to this extended SFR space. However, instructions to access the extended SFR area are 1 byte longer than instructions to access the SFR area.

(6) External memory space

The external memory space is space that can be accessed by setting the memory expansion mode register. This memory space is extends up from flash memory to EDFFFH.

In separate mode, 28 external pins (A19 to A0 and D7 to D0) are available. In multiplexed mode, 20 external pins (A19 to A0,D7 to D0) area available.

For pin settings when using external memory, refer to the chapter describing port functions in the user's manual of the target device.

Caution To fetch instructions from the external memory area, start with a branch instruction (CALL or BR) in flash or RAM memory and end with a return instruction (RET, RETB, or RETI) in the external memory area.

Although the flash memory area is adjacent to the external memory area, programs cannot be located so as to straddle these two areas.

4.6.3 Registers

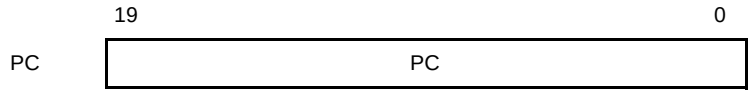
(1) Control registers

Control registers are registers with special functions for controlling the program sequence, program status, and stack memory. They include the program counter, the program status word, and the stack pointer.

(a) Program counter (PC)

The program counter is a 20-bit register that holds the address of the next program to be executed.

Figure 4-10. Configuration of Program Counter



(b) Program status word (PSW)

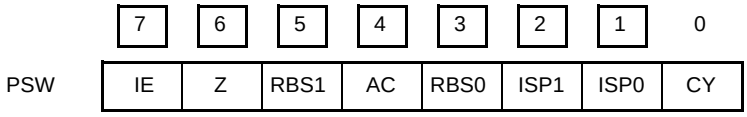
The program status word is an 8-bit register consisting of flags that are set and reset by instruction execution.

The ISP1 flag is added as bit 2 in products that support 4 interrupt levels.

The program status word is automatically saved on the stack when an interrupt request occurs and a PUSH PSW instruction is executed, and is automatically restored when an RETB or RETI instruction and a POP PSW instruction are executed.

The PSW is set to 06H on reset signal input.

Figure 4-11. Configuration of Program Status Word



- Interrupt enable flag (IE)

  This flag controls interrupt request acknowledgement by the CPU.

  When IE = 0, interrupts are disabled (DI), and interrupts other than non-maskable interrupts are all disabled.

  When IE = 1, interrupts are enabled (EI), and interrupt request acknowledgement is controlled by the interrupt mask flags for the various interrupt sources, and by the priority specification flags.

  This flag is reset (0) by execution of the DI instruction or by interrupt request acknowledgment. It is set (1) by execution of the EI instruction.

- Zero flag (Z)

  This flag is set (1) when the result of an operation is zero. It is reset (0) in all other cases.

- Register bank selection flags (RBS0,RBS1)

  These are two 1-bit flags used to select one of the 4 register banks.

  The 2 bits of information in these flags indicate the bank selected by execution of an SBL RBn instruction.

- Auxiliary carry flag (AC)

  This flag is set (1) if an operation has a carry from bit 3 or a borrow to bit 3. It is reset (0) in all other cases.

- In-service priority flags (ISP0 and ISP1)

  These flags manage the priority of acknowledgeable maskable vectored interrupts. Acknowledgment is disabled for vectored interrupt requests with priorities lower than the ISP0 and ISP1 values, as specified by the priority specification flag registers (PR). Actual acknowledgment for interrupt requests is controlled by the state of the interrupt enable flag (IE).

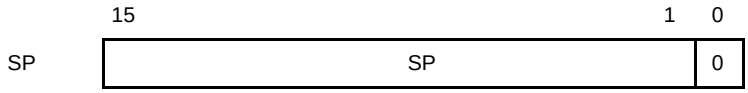
- Carry flag (CY)

  This flag stores overflow or underflow on execution of an add/subtract instruction. It stores the shift-out value on execution of a rotate instruction, and functions as a bit accumulator during execution of bit manipulation instructions.

(c) Stack pointer (SP)

This is a 16-bit register that holds the start address of the stack. The stack can be located in internal RAM.

Figure 4-12. Configuration of Stack Pointer



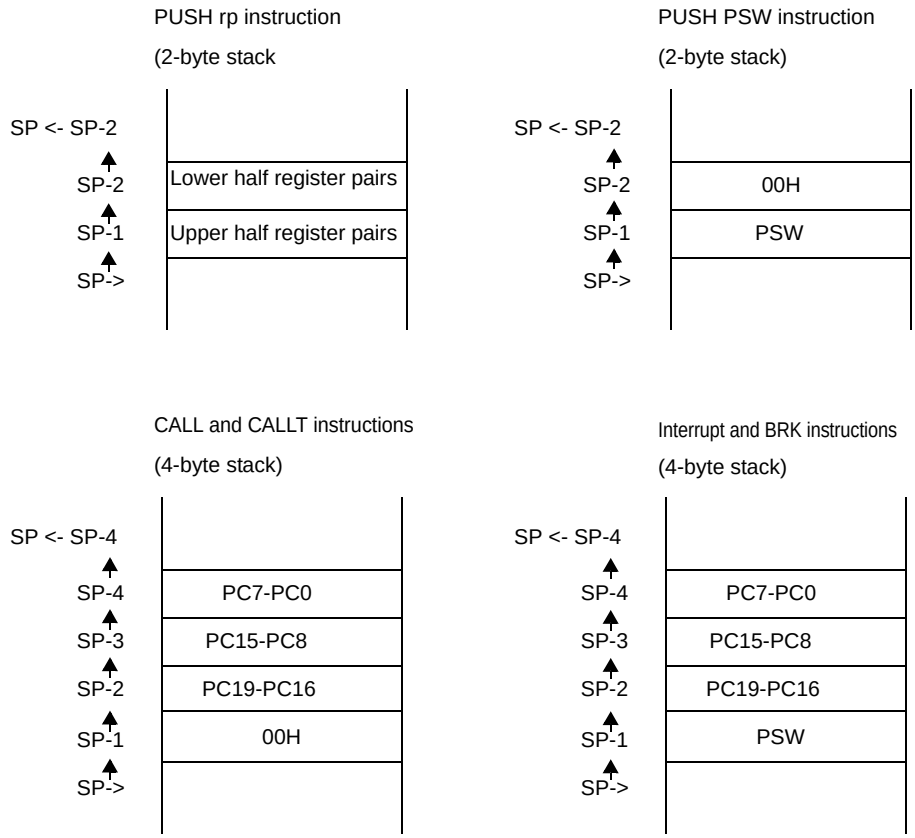
SP is decremented before write (save) to the stack and is incremented after read (restored) from the stack. RESET input leaves the contents of SP undefined, so be sure to initialize SP before instruction execution. Always specify the SP address as an even address. If an odd address is specified, the LSB is set to fixed 0. Because the memory space of 78K0R microcontrollers has been expanded, stack addresses used for call instructions and interrupts are 1 byte longer. Due to the 16-bit width of stack RAM, the stack data size is 2 bytes or 4 bytes. (See the following "Table 4-24. Stack Data Size Differences Between 78K0 and 78K0R Microcontrollers".)

Table 4-24. Stack Data Size Differences Between 78K0 and 78K0R Microcontrollers

Save Instruction	Restore Instruction	Stack Data Sizes of 78K0 Microcontrollers	Stack Data Sizes of 78K0R Microcontrollers
PUSH rp	POP rp	2 bytes	2 bytes
PUSH PSW	POP PSW	1 byte	2 bytes
CALL, CALLT	RET	2 bytes	4 bytes
Interrupt	RETI	3 bytes	4 bytes
BRK	RETB	3 bytes	4 bytes

The following figure shows the data savedby stack operations in 78K0R microcontrollers.

Figure 4-13. Data Saved to Stack Memory



The stack pointer may point to internal RAM only. It is possible to specify addresses in the range F0000H to FFFFFH, so be careful not to exceed the memory space of internal RAM. If an address outside the internal RAM space is specified, write operations to that address are ignored and read operations return undefined values.

(2) General registers

On-chip general registers are mapped to RAM addresses FFEE0H to FFEFFH. There are 4 register banks, each bank consisting of eight 8-bit registers (X, A, C, B, E, D, L and H). The CPU control instruction "SEL RBn" selects the bank to be used in instruction execution.

Each register can be used as an 8-bit register, and register pairs of two 8-bit registers can be used as 16-bit registers.

Programs can specify registers by their function names (X, A, C, B, E, D, L, H, AX, BC, DE, HL) or by their absolute names (R0 to R7, RP0 to RP3).

Caution Use of the general register space (FFEE0H to FFEFFH) as an instruction fetch area or stack area is prohibited.

Table 4-25. List of General-purpose Registers (78K0 Compatible)

Bank Name	Register				Absolute Address
	Function Name		Absolute Name		
	16-bit	8-bit	16-bit	8-bit	
BANK0	HL	H	RP3	R7	FFEFFH
		L		R6	FFEFEH
	DE	D	RP2	R5	FFEFDH
		E		R4	FFEFCH
	BC	B	RP1	R3	FFEFBH
		C		R2	FFEFAH
	AX	A	RP0	R1	FFEF9H
		X		R0	FFEF8H
BANK1	HL	H	RP3	R7	FFEF7H
		L		R6	FFEF6H
	DE	D	RP2	R5	FFEF5H
		E		R4	FFEF4H
	BC	B	RP1	R3	FFEF3H
		C		R2	FFEF2H
	AX	A	RP0	R1	FFEF1H
		X		R0	FFEF0H
BANK2	HL	H	RP3	R7	FFEEFH
		L		R6	FFEEEH
	DE	D	RP2	R5	FFEEDH
		E		R4	FFEECH
	BC	B	RP1	R3	FFEEBH
		C		R2	FFEEAH
	AX	A	RP0	R1	FFEE9H
		X		R0	FFEF8H
BANK3	HL	H	RP3	R7	FFEE7H
		L		R6	FFEE6H
	DE	D	RP2	R5	FFEE5H
		E		R4	FFEE4H
	BC	B	RP1	R3	FFEE3H
		C		R2	FFEE2H
	AX	A	RP0	R1	FFEE1H
		X		R0	FFEE0H

(3) ES and CS registers

The ES and CS registers were added for the 78K0R microcontrollers. The ES register specifies the high-order address byte for data instructions, and the CS register specifies the high-order address byte for branch instructions. See "(2) Addressing of data addresses" for more information about how to use the ES register, and see "(1) Addressing of instruction addresses" for more information about how to use the CS register.

On reset, ES is set to 0FH and CS is set to 00H

Figure 4-14. Configurations of ES and CS Registers

	7	6	5	4	3	2	1	0
ES	0	0	0	0	ES3	ES2	ES1	ES0

	7	6	5	4	3	2	1	0
CS	0	0	0	0	CS3	CP2	CP1	CP0

(4) Special function registers (SFR)

The following table lists the fixed-address SFRs of 78K0R microcontrollers.

Table 4-26. List of Fixed-address SFRs

Address	Register Name
FFFF8H	SPL
FFFF9H	SPH
FFFFAH	PSW
FFFFBH	Reserve
FFFFCH	CS
FFFFDH	ES
FFFFEH	PMC
FFFFFH	MEM

(a) Processor mode control register (PMC)

This is an 8-bit register for control of processor modes. For details, see "(2) Internal program memory space".

On reset, PMC is set to 00H.

Figure 4-15. Configuration of Processor Mode Control Register

Address:FFFFEH On reset:00H R/W

Symbol	7	6	5	4	3	2	1	0
PMC	0	0	0	0	0	0	0	MAA

MAA	Flash Memory Space Mirrored to Area F0000H to FFFFFH Note
0	00000H to 0FFFFH is mirrored to F0000H to FFFFFH
1	10000H to 1FFFFH is mirrored to F0000H to FFFFFH

Note SFR and RAM areas are also allocated to the range from F0000H to FFFFFH. In areas of overlap, they take priority.

Cautions 1. The processor mode control should be set once only, when it is first initialized. After initialization, writing to PMC is prohibited.

2. After setting PMC, wait for at least one instruction before accessing the mirror area.

4.6.4 Addressing

There are two types of addressing: addressing of data addresses, and addressing of program addresses. This section describes the addressing modes of both types of addressing.

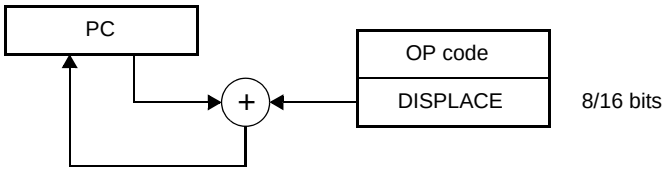
(1) Addressing of instruction addresses

(a) Relative addressing

Relative addressing adds a displacement value from the instruction word (signed complement data: -128 to +127 or -32768 to +32767) to the value in the program counter (PC), and stores the sum in the program counter. Execution branches to the specified program address.

Relative addressing is applied only to branch instructions.

Figure 4-16. How Relative Addressing Works



(b) Immediate addressing

Immediate addressing specifies a branch destination program address by storing immediate data from the program word in the program counter.

There are two types of Immediate addressing: CALL !!addr20 or BR !!addr20 specifies 20-bit addresses, and CALL !addr16 or BR !addr16 specifies 16-bit addresses. When a 16-bit address is specified, 0000 is set in the high-order 4 bits.

Figure 4-17. Example of CALL !!addr20/BR !!addr20 Addressing

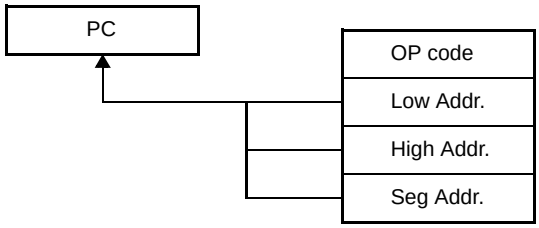
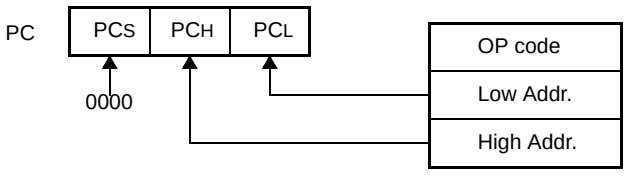


Figure 4-18. Example of CALL !addr16/BR !addr16 Addressing

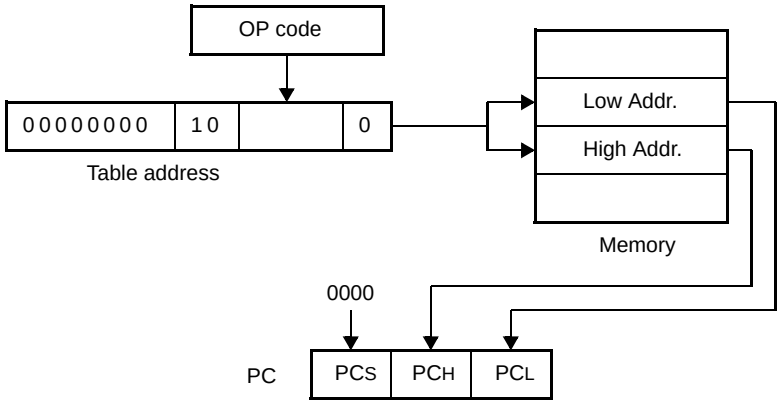


(c) Table indirect addressing

Table indirect addressing specifies an address in the CALLT table area (0080H to 00BFH) with 5 bits of immediate data in the instruction word. Then it stores the contents of that CALLT table address, and the next CALLT table address, as 16-bit data in the program counter. This specifies a program address to be called. Table indirect addressing is applied only to the CALLT instruction.

In 78K0R microcontrollers, branching is enabled only to the 64 KB space from 00000H to 0FFFFH.

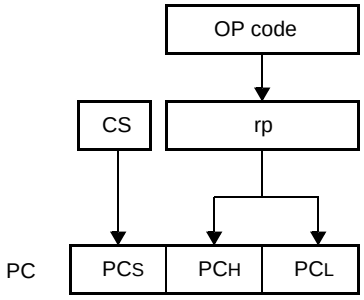
Figure 4-19. How Table Indirect Addressing Works



(d) Register direct addressing

In register direct addressing, the program instruction word specifies a general-purpose register pair (AX/BC/DE/HL) in the current register bank. The content of that register pair and the current CS register is then stored in the program counter as 20-bit data. This specifies a call or branch program address. Register direct addressing is applied only to the CALL AX/BC/DE/HL instructions and the BR AX instruction.

Figure 4-20. How register Direct Addressing Works



(2) Addressing of data addresses

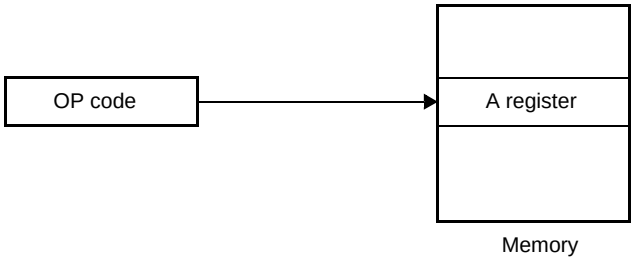
(a) Implied addressing

Implied addressing is used by instructions that access a register with a special function, such as the accumulator. The instruction word contains no special field to specify a register. The register specification is implied by the instruction word itself.

Because register specification is implied, the instruction has no operand.

In 78K0R microcontrollers, implied addressing is applied to the MULU X instruction only.

Figure 4-21. How Implied Addressing Works



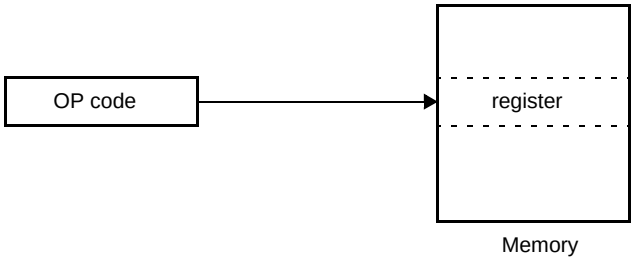
(b) Register addressing

Register addresses accesses memory using a general register an operand. It uses 3 bits in the instruction word to specify an 8-bit register, or 2 bits to specify a 16-bit register.

The operand format is shown below.

Format	Description
r	X, A, C, B, E, D, L, H
rp	AX, BC, DE, HL

Figure 4-22. How Register Addressing Works



(c) Direct addressing

Direct addressing uses immediate data in the instruction word as the operand. It specifies the target address directly.

The operand format is shown below.

Format	Description
ADDR16	Label, or 16-bit immediate data (F0000H to FFFFFH only)
ES:ADDR16	Label, or 16-bit immediate data (high-order 4 bits specified by ES register)

Figure 4-23. Example of ADDR16 Addressing

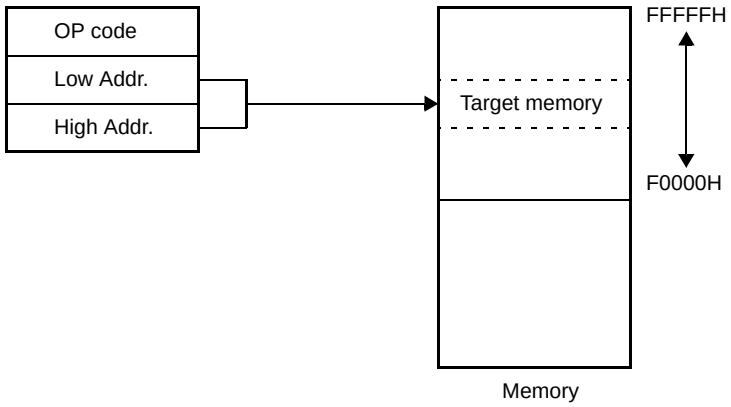
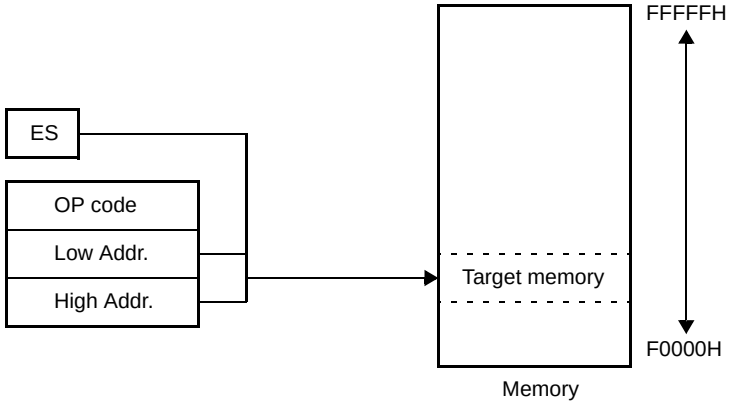


Figure 4-24. Example of ES:ADDR16 Addressing



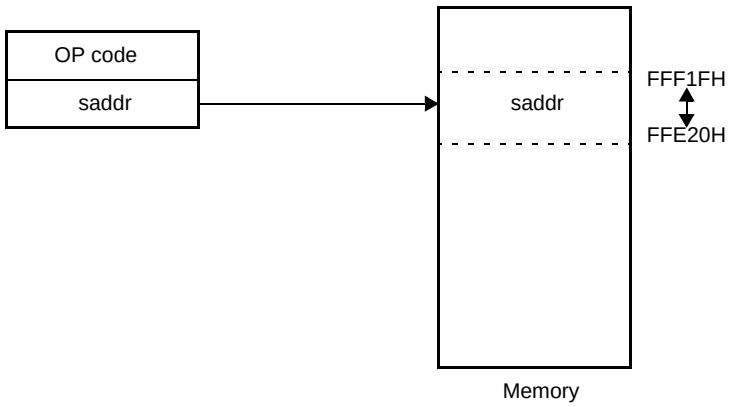
(d) Short direct addressing

Short direct addressing specifies the target address directly using 8 bits of data in the instruction word. This type of addressing is applied only to the space from FFE20H to FFF1FH.

The operand format is shown below.

Format	Description
SADDR	Label, or immediate data for FFE20H to FFF1FH, or immediate data for 0FE20H to 0FF1FH (Limited to space from FFE20H to FFF1FH)
SADDRP	Label, or immediate data for FFE20H to FFF1FH, or immediate data for 0FE20H to 0FF1FH (even addresses only) (Limited to space from FFE20H to FFF1FH)

Figure 4-25. How Short Direct Addressing Works



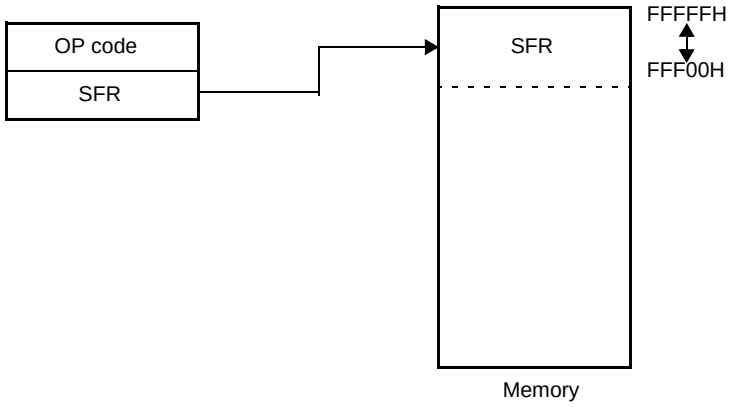
Remark A SADDR or SADDRP specification (omitting the high-order 4 bits of the address) can specify the values FE20H to FF1FH as 16 bits of immediate data or the values FFE20H to FFF1FH as 20 bits of immediate data.
Regardless of which format is used, the specified addresses are those in the memory space FFE20H to FFF1FH.

(e) SFR addressing

SFR addressing specifies a target SFR address directly using 8 bits of data in the instruction word. This type of addressing is applied only to the space from FFF00H to FFFFFH.
The operand format is shown below.

Format	Description
SFR	SFR register name
SFRP	16-bit SFR register name (even address only)

Figure 4-26. How SFR Addressing Works



(f) Register indirect addressing

Register indirect addressing uses data in the instruction word to specify a register pair. The contents of the specified register pair are then used as the operand to specify the target memory address.

The operand format is shown below.

Format	Description
-	[DE], [HL] (F0000H to FFFFFH only)
-	ES:[DE], ES:[HL] (high-order 4 bits of address specified by ES register)

Figure 4-27. Example of [DE] and [HL] Addressing

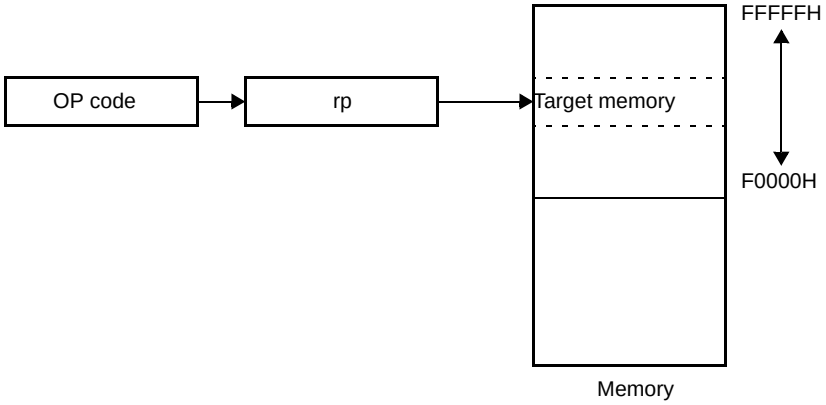
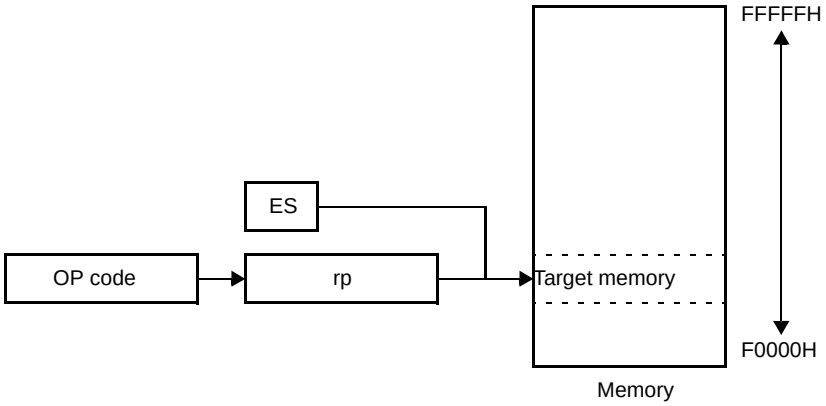


Figure 4-28. Example of ES:[DE] and ES:[HL] Addressing



(g) Based addressing

In based addressing, the instruction word specifies a register pair and an offset. The offset (8-bit or 16-bit immediate data) is added to the contents of the base register pair to specify the target address.
The operand format is shown below.

Format	Description
-	[HL + byte], [DE + byte], [SP + byte] (F0000H to FFFFFH only)
-	word[B], word[C] (F0000H to FFFFFH only)
-	word[BC] (F0000H to FFFFFH only)
-	ES:[HL + byte], ES:[DE + byte] (high-order 4 bits of address specified by ES register)
-	ES:word[B], ES:word[C] (high-order 4 bits of address specified by ES register)
-	ES:word[BC] (high-order 4 bits of address specified by ES register)

Figure 4-29. Example of [SP+byte] Addressing

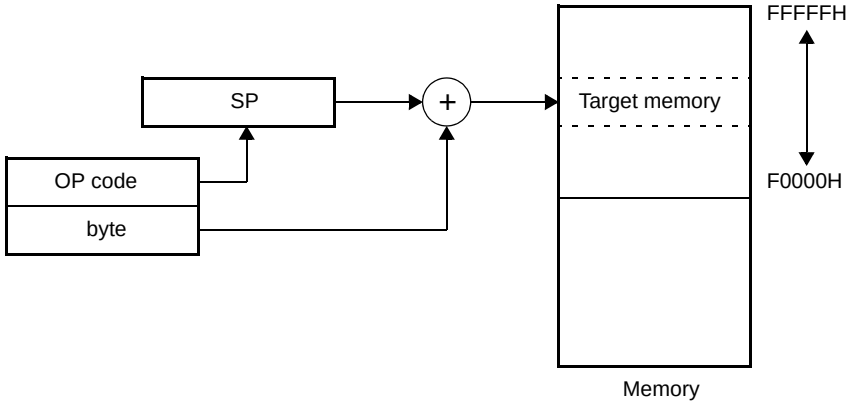


Figure 4-30. Example of [HL+byte] and [DE+byte] Addressing

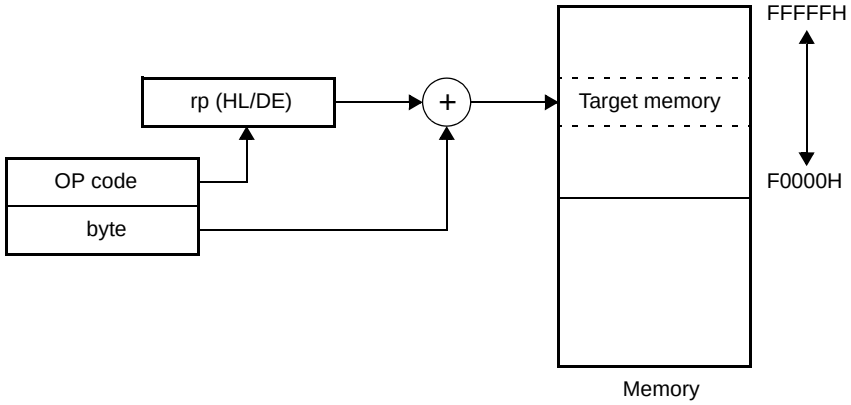


Figure 4-31. Example of word[B] and word[C] Addressing

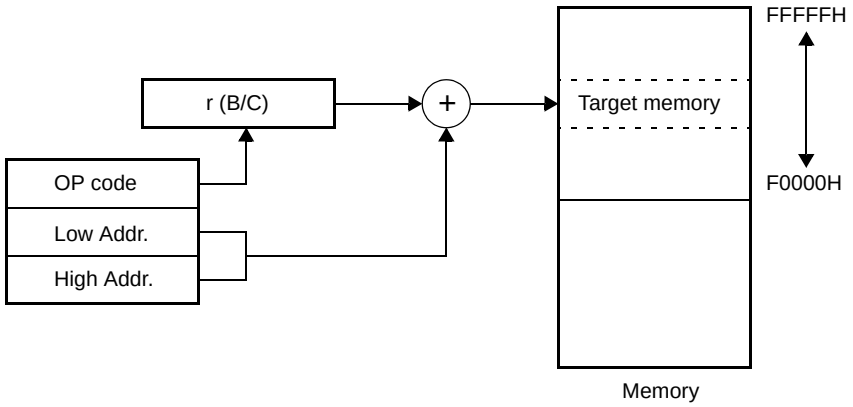


Figure 4-32. Example of word[BC] Addressing

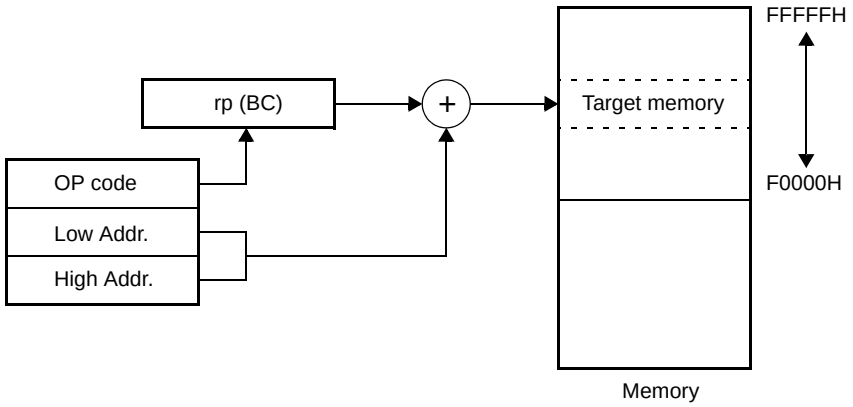


Figure 4-33. Example of ES:[HL+byte] and ES:[DE+byte] Addressing

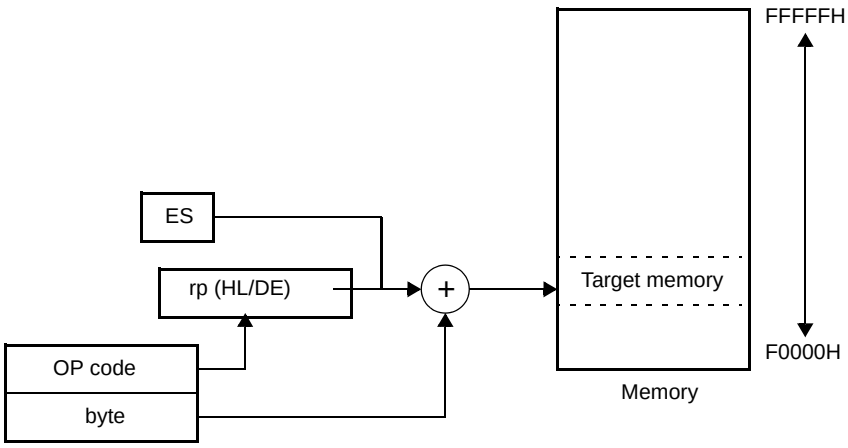


Figure 4-34. Example of ES:word[B] and ES:word[C] Addressing

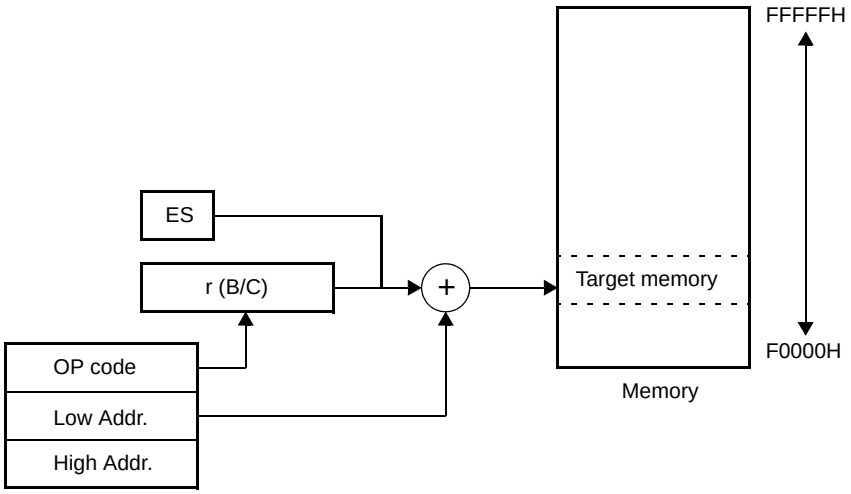
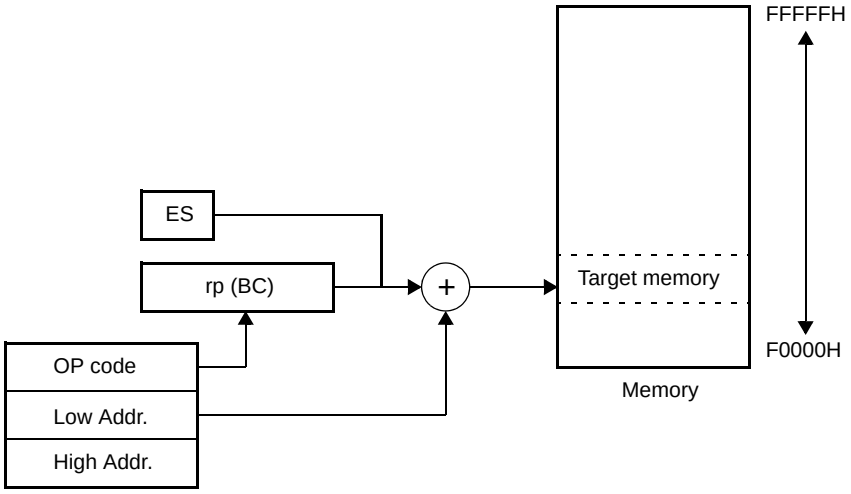


Figure 4-35. Example of ES:word[BC] Addressing



(h) Based indexed addressing

In based index addressing, the instruction word specifies a register pair as a base register and either the B or C register as an offset register. The contents of the base register are added to the contents of the offset register to specify the target address.

The operand format is shown below.

Format	Description
-	[HL + B], [HL + C] (F0000H to FFFFFH only)
-	ES:[HL + B], ES:[HL + C] (high-order 4 bits of address specified by ES register)

Figure 4-36. Example of [HL+B] and [HL+C] Addressing

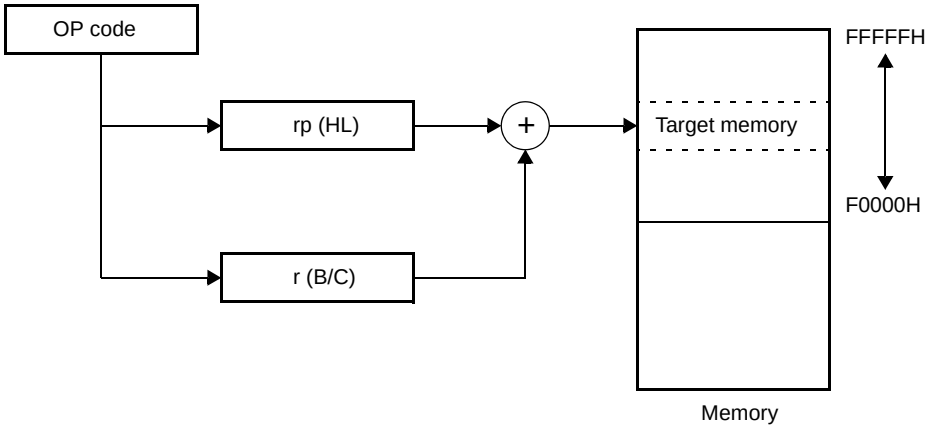
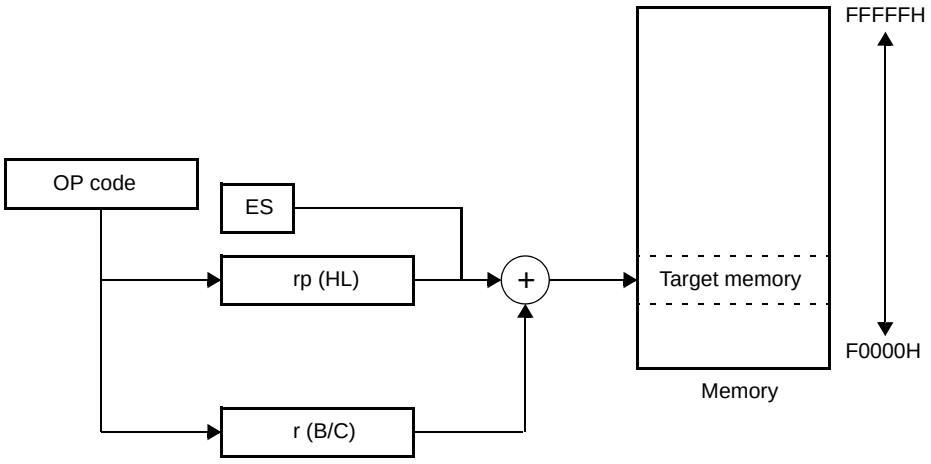


Figure 4-37. Example of ES:[HL+B] and ES:[HL+C] Addressing



(i) Stack addressing

Stack addressing accesses the stack indirectly using the contents of the stack pointer (SP). This type of addressing is employed automatically when the PUSH and POP instructions are executed, when subroutine call and return instructions are executed, and when registers are saved and restored upon generation of an interrupt request.

Stack addressing is applied only to the internal RAM area.

The operand format is shown below.

Format	Description
-	PUSH AX/BC/DE/HL POP AX/BC/DE/HL CALL/CALLT RET BRK RETB (interrupt request generated) RETI

4.6.5 Instruction set

This chapter lists the instructions of the 78K0R microcontroller instruction set.
These instructions are common to all microcontrollers in the 78K0R microcontroller.

(1) Expressive form of the operand and description method

The operands in the "Operands" field of each instruction are shown in the representation for that type of operand. (For details, see the assembler specifications.) When there are two or more ways to specify an operand in source code, select one of them. Alphabetic characters written by a capital letter, the symbols #, !, !!, $, $!, [ ], and ES: are keywords and should be written just as they appear. Symbols have the following meanings.

#	Immediate data specification
!	16-bit absolute address specification
!!	20-bit absolute address specification
$	8-bit relative address specification
$!	16-bit relative address specification
[ ]	Indirect address specificatio
ES	Extension address specification

Specify immediate data with an appropriate value or label. When specifying a label, be sure to include the #, !, !!, $, $!, [ ], or ES: symbol.
For register operands, r and rp can be replaced by register function names (X, A, C, etc.) or register absolute names (R0, R1, R2, etc., as shown in parentheses in the following table).

Table 4-27. Operand Type Representations and Source Code Formats

Format	Description
r	X(R0), A(R1), C(R2), B(R3), E(R4), D(R5), L(R6), H(R7)
rp	AX(RP0), BC(RP1), DE(RP2), HL(RP3)
sfr	Special-function register name (SFR name)
sfrp	Special-function register name (16-bit SFR, even addresses only[Note])
saddr	FFE20H to FFF1FH : Immediate data or label
saddrp	FFE20H to FFF1FH : Immediate data or labels (even addresses only[Note])
addr20	00000H to FFFFFH : Immediate data or label
addr16	0000H to 0FFFFH : Immediate data or label (even addresses only for 16-bit data transfer instructions[Note])
addr5	0080H to 00BFH : Immediate data or label (even addresses only)
word	16-bit immediate data or label
byte	8-bit immediate data or label
bit	3-bit immediate data or label
RBn	RB0, RB1, RB2, RB3

Note Bit 0 = 0 when an odd address is specified.

(2) Operation field symbols

The "Operation" field uses the following symbols to designate the operation that occurs when the instruction is executed.

Table 4-28. Operation Field Symbols

Symbol	Function
A	A register:8-bit accumulator
X	X register
B	B register
C	C register
D	D register
E	E register
H	H register
L	L register
ES	ES register
CS	CS register
AX	AX register pair:16-bit accumulator
BC	BC register pair
DE	DE register pair
HL	HL register pair
PC	Program counter
SP	Stack pointer
PSW	Program status word
CY	Carry flag
AC	Auxiliary carry flag
Z	Zero flag
RBS	Register bank selection flag
IE	Interrupt request enable flag
()	Memory contents indicated by address or register contents in parentheses
XH, XL XS, XH, XL	16-bit registers: XH = high-order 8 bits, XL = low-order 8 bits 20-bit registers: XS = bits 19 to 16, XH = bits 15 to 8, XL = bits 7 to 0
^	Logical AND
v	Logical OR
∀	Exclusive OR
—	Inverted data
addr16	16-bit immediate data
addr20	20-bit immediate data
jdisp8	Signed 8-bit data (displacement value)
jdisp16	Signed 16-bit data (displacement value)

(3) Flag field symbols

The "Flag" field uses the following symbols to designate flag changes that occur when the instruction is executed.

Table 4-29. Flag Field Symbols

Symbol	Flag Change
(Blank)	Unchanged
0	Cleared to 0
1	Set to 1
x	Set or cleared according to the result
R	Previously saved value is restored

(4) PREFIX instructions

Some instructions are shown with the ES: prefix. The addition of the prefix makes it possible to expand the accessible data space from the 64 KB space [F0000H to FFFFFH] to the 1 MB space [00000H to FFFFFH]. This is done by adding the value of the ES register to the address specification. When a PREFIX operation code is attached as a prefix to the target instruction, only one instruction immediately after the PREFIX operation code is executed as the addresses with the ES register value added.

Table 4-30. Examples of PREFIX Instructions in Use

Instruction	Opcode				
	1	2	3	4	5
MOV !addr16, #byte	CFH	!addr16		#byte	-
MOV ES:!addr16, #byte	11H	CFH	!addr16		#byte
MOV A, [HL]	8BH	-	-	-	-
MOV A, ES:[HL]	11H	8BH	-	-	-

Caution Before executing a PREFIX instruction, always set the correct value in the ES register, for example with MOV ES, A.

(5) Operation list

(a) 8-bit data transfer instructions

Table 4-31. Operation List (8-bit Data Transfer Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
MOV	r, #byte	2	1	-	r <- byte			
	saddr, #byte	3	1	-	(saddr) <- byte			
	sfr, #byte	3	1	-	sfr <- byte			
	!addr16, #byte	4	1	-	(addr16) <- byte			
	A, r [Note 3]	1	1	-	A <- r			
	r, A [Note 3]	1	1	-	r <- A			
	A, saddr	2	1	-	A <- (saddr)			
	saddr, A	2	1	-	(saddr) <- A			
	A, sfr	2	1	-	A <- sfr			
	sfr, A	2	1	-	sfr <- A			
	A, !addr16	3	1	4	A <- (addr16)			
	!addr16, A	3	1	-	(addr16) <- A			
	PSW, #byte	3	3	-	PSW <- byte	x	x	x
	A, PSW	2	1	-	A <- PSW			
	PSW, A	2	3	-	PSW <- A	x	x	x
	ES, #byte	2	1	-	ES <- byte			
	ES, saddr	3	1	-	ES <- (saddr)			
	A, ES	2	1	-	A <- ES			
	ES, A	2	1	-	ES <- A			
	CS, #byte	3	1	-	CS <- byte			
	A, CS	2	1	-	A <- CS			
	CS, A	2	1	-	CS <- A			
	A, [DE]	1	1	4	A <- (DE)			
	[DE], A	1	1	-	(DE) <- A			
	[DE+byte], #byte	3	1	-	(DE + byte) <- byte			
	A, [DE+byte]	2	1	4	A <- (DE + byte)			
	[DE+byte], A	2	1	-	(DE + byte) <- A			
	A, [HL]	1	1	4	A <- (HL)			
	[HL], A	1	1	-	(HL) <- A			
	[HL+byte], #byte	3	1	-	(HL + byte) <- byte			
	A, [HL+byte]	2	1	4	A <- (HL + byte)			
	[HL+byte], A	2	1	-	(HL + byte) <- A			
	A, [HL+B]	2	1	4	A <- (HL + B)			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
	[HL+B], A	2	1	-	(HL + B) <- A			
	A, [HL+C]	2	1	4	A <- (HL + C)			
	[HL+C], A	2	1	-	(HL + C) <- A			
	word[B], #byte	4	1	-	(B + word) <- byte			
	A, word[B]	3	1	4	A <- (B + word)			
	word[B], A	3	1	-	(B + word) <- A			
	word[C], #byte	4	1	-	(C + word) <- byte			
	A, word[C]	3	1	4	A <- (C + word)			
	word[C], A	3	1	-	(C + word) <- A			
	word[BC], #byte	4	1	-	(BC + word) <- byte			
	A, word[BC]	3	1	4	A <- (BC + word)			
	word[BC], A	3	1	-	(BC + word) <- A			
	[SP+byte], #byte	3	1	-	(SP + byte) <- byte			
	A, [SP+byte]	2	1	-	A <- (SP + byte)			
	[SP+byte], A	2	1	-	(SP + byte) <- A			
	B, saddr	2	1	-	B <- (saddr)			
	B, !addr16	3	1	4	B <- (addr16)			
	C, saddr	2	1	-	C <- (saddr)			
	C, !addr16	3	1	4	C <- (addr16)			
	X, saddr	2	1	-	X <- (saddr)			
	X, !addr16	3	1	4	X <- (addr16)			
	ES:!addr16, #byte	5	2	-	(ES, addr16) <- byte			
	A, ES:!addr16	4	2	5	A <- (ES, addr16)			
	ES:!addr16, A	4	2	-	(ES, addr16) <- A			
	A, ES:[DE]	2	2	5	A <- (ES, DE)			
	ES:[DE], A	2	2	-	(ES, DE) <- A			
	ES:[DE+byte], #byte	4	2	-	((ES, DE) + byte) <- byte			
	A, ES:[DE+byte]	3	2	5	A <- ((ES, DE) + byte)			
	ES:[DE+byte], A	3	2	-	((ES, DE) + byte) <- A			
	A, ES:[HL]	2	2	5	A <- (ES, HL)			
	ES:[HL], A	2	2	-	(ES, HL) <- A			
	ES:[HL+byte], #byte	4	2	-	((ES, HL) + byte) <- byte			
	A, ES:[HL+byte]	3	2	5	A <- ((ES, HL) + byte)			
	ES:[HL+byte], A	3	2	-	((ES, HL) + byte) <- A			
	A, ES:[HL+B]	3	2	5	A <- ((ES, HL) + B)			
	ES:[HL+B], A	3	2	-	((ES, HL) + B) <- A			
	A, ES:[HL+C]	3	2	5	A <- ((ES, HL) + C)			
	ES:[HL+C], A	3	2	-	((ES, HL) + C) <- A			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
	ES:word[B], #byte	5	2	-	((ES, B) + word) <- byte			
	A, ES:word[B]	4	2	5	A <- ((ES, B) + word)			
	ES:word[B], A	4	2	-	((ES, B) + word) <- A			
	ES:word[C], #byte	5	2	-	((ES, C) + word) <- byte			
	A, ES:word[C]	4	2	5	A <- ((ES, C) + word)			
	ES:word[C], A	4	2	-	((ES, C) + word) <- A			
	ES:word[BC], #byte	5	2	-	((ES, BC) + word) <- byte			
	A, ES:word[BC]	4	2	5	A <- ((ES, BC) + word)			
	ES:word[BC], A	4	2	-	((ES, BC) + word) <- A			
	B, ES:!addr16	4	2	5	B <- (ES, addr16)			
	C, ES:!addr16	4	2	5	C <- (ES, addr16)			
	X, ES:!addr16	4	2	5	X <- (ES, addr16)			
XCH	A, r [Note 3]	1 (r = X) 2 (except r = X)	1	-	A <--> r			
	A, saddr	3	2	-	A <--> (saddr)			
	A, sfr	3	2	-	A <--> sfr			
	A, !addr16	4	2	-	A <--> (addr16)			
	A, [DE]	2	2	-	A <--> (DE)			
	A, [DE+byte]	3	2	-	A <--> (DE + byte)			
	A, [HL]	2	2	-	A <--> (HL)			
	A, [HL+byte]	3	2	-	A <--> (HL + byte)			
	A, [HL+B]	2	2	-	A <--> (HL + B)			
	A, [HL+C]	2	2	-	A <--> (HL + C)			
	A, ES:!addr16	5	3	-	A <--> (ES, addr16)			
	A, ES:[DE]	3	3	-	A <--> (ES, DE)			
	A, ES:[DE+byte]	4	3	-	A <--> ((ES, DE) + byte)			
	A, ES:[HL]	3	3	-	A <--> (ES, HL)			
	A, ES:[HL+byte]	4	3	-	A <--> ((ES, HL) + byte)			
	A, ES:[HL+B]	3	3	-	A <--> ((ES, HL) + B)			
	A, ES:[HL+C]	3	3	-	A <--> ((ES, HL) + C)			
	A	1	1	-	A <- 01H			
	X	1	1	-	X <- 01H			
	B	1	1	-	B <- 01H			
	C	1	1	-	C <- 01H			
	saddr	2	1	-	(saddr) <- 01H			
	!addr16	3	1	-	(addr16) <- 01H			
	ES:!addr16	4	2	-	(ES, addr16) <- 01H			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
CLRB	A	1	1	-	A <- 00H			
	X	1	1	-	X <- 00H			
	B	1	1	-	B <- 00H			
	C	1	1	-	C <- 00H			
	saddr	2	1	-	(saddr) <- 00H			
	!addr16	3	1	-	(addr16) <- 00H			
	ES:!addr16	4	2	-	(ES,addr16) <- 00H			
MOVS	[HL+byte], X	3	1	-	(HL + byte) <- X	x		x
	ES:[HL+byte], X	4	2	-	(ES, HL + byte) <- X	x		x

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. Except r = A.

4. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(b) 16-bit data transfer instructions

Table 4-32. Operation List (16-bit Data Transfer Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
MOVW	rp, #word	3	1	-	rp <- word			
	saddrp, #word	4	1	-	(saddrp) <- word			
	sfrp, #word	4	1	-	sfrp <- word			
	AX, saddrp	2	1	-	AX <- (saddrp)			
	saddrp, AX	2	1	-	(saddrp) <- AX			
	AX, sfrp	2	1	-	AX <- sfrp			
	sfrp, AX	2	1	-	sfrp <- AX			
	AX, rp [Note 3]	1	1	-	AX <- rp			
	rp, AX [Note 3]	1	1	-	rp <- AX			
	AX, !addr16	3	1	4	AX <- (addr16)			
	!addr16, AX	3	1	-	(addr16) <- AX			
	AX, [DE]	1	1	4	AX <- (DE)			
	[DE], AX	1	1	-	(DE) <- AX			
	AX, [DE+byte]	2	1	4	AX <- (DE + byte)			
	[DE+byte], AX	2	1	-	(DE + byte) <- AX			
	AX, [HL]	1	1	4	AX <- (HL)			
	[HL], AX	1	1	-	(HL) <- AX			
	AX, [HL+byte]	2	1	4	AX <- (HL + byte)			
	[HL+byte], AX	2	1	-	(HL + byte) <- AX			
	AX, word[B]	3	1	4	AX <- (B + word)			
	word[B], AX	3	1	-	(B + word) <- AX			
	AX, word[C]	3	1	4	AX <- (C + word)			
	word[C], AX	3	1	-	(C + word) <- AX			
	AX, word[BC]	3	1	4	AX <- (BC + word)			
	word[BC], AX	3	1	-	(BC + word) <- AX			
	AX, [SP+byte]	2	1	-	AX <- (SP + byte)			
	[SP+byte], AX	2	1	-	(SP + byte) <- AX			
	BC, saddrp	2	1	-	BC <- (saddrp)			
	BC, !addr16	3	1	4	BC <- (addr16)			
	DE, saddrp	2	1	-	DE <- (saddrp)			
	DE, !addr16	3	1	4	DE <- (addr16)			
	HL, saddrp	2	1	-	HL <- (saddrp)			
	HL, !addr16	3	1	4	HL <- (addr16)			
	AX, ES:!addr16	4	2	5	AX <- (ES, addr16)			
	ES:!addr16, AX	4	2	-	(ES, addr16) <- AX			
	AX, ES:[DE]	2	2	5	AX <- (ES, DE)			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
	ES:[DE], AX	2	2	-	(ES, DE) <- AX			
	AX, ES:[DE+byte]	3	2	5	AX <- ((ES, DE) + byte)			
	ES:[DE+byte], AX	3	2	-	((ES, DE) + byte) <- AX			
	AX, ES:[HL]	2	2	5	AX <- (ES, HL)			
	ES:[HL], AX	2	2	-	(ES, HL) <- AX			
	AX, ES:[HL+byte]	3	2	5	AX <- ((ES, HL) + byte)			
	ES:[HL+byte], AX	3	2	-	((ES, HL) + byte) <- AX			
	AX, ES:word[B]	4	2	5	AX <- ((ES, B) + word)			
	ES:word[B], AX	4	2	-	((ES, B) + word) <- AX			
	AX, ES:word[C]	4	2	5	AX <- ((ES, C) + word)			
	ES:word[C], AX	4	2	-	((ES, C) + word) <- AX			
	AX, ES:word[BC]	4	2	5	AX <- ((ES, BC) + word)			
	ES:word[BC], AX	4	2	-	((ES, BC) + word) <- AX			
	BC, ES:!addr16	4	2	5	BC <- (ES, addr16)			
	DE, ES:!addr16	4	2	5	DE <- (ES, addr16)			
	HL, ES:!addr16	4	2	5	HL <- (ES, addr16)			
XCHW	AX, rp [Note 3]	1	1	-	AX <--> rp			
ONEW	AX	1	1	-	AX <- 0001H			
	BC	1	1	-	BC <- 0001H			
CLRW	AX	1	1	-	AX <- 0000H			
	BC	1	1	-	BC <- 0000H			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. Except rp = AX

4. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(c) 8-bit manipulation instructions

Table 4-33. Operation List (8-bit Manipulation Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
ADD	A, #byte	2	1	-	A, CY <- A + byte	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) + byte	x	x	x
	A, r [Note 3]	2	1	-	A, CY <- A + r	x	x	x
	r, A	2	1	-	r, CY <- r + A	x	x	x
	A, saddr	2	1	-	A, CY <- A + (saddr)	x	x	x
	A, !addr16	3	1	4	A, CY <- A + (addr16)	x	x	x
	A, [HL]	1	1	4	A, CY <- A + (HL)	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A + (HL + byte)	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A + (HL + B)	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A + (HL + C)	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A + (ES, addr16)	x	x	x
	A, ES:[HL]	2	2	5	A,CY <- A + (ES, HL)	x	x	x
	A, ES:[HL+byte]	3	2	5	A,CY <- A + ((ES, HL) + byte)	x	x	x
	A, ES:[HL+B]	3	2	5	A,CY <- A + ((ES, HL) + B)	x	x	x
	A, ES:[HL+C]	3	2	5	A,CY <- A + ((ES, HL) + C)	x	x	x
ADDC	A, #byte	2	1	-	A, CY <- A + byte + CY	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) + byte + CY	x	x	x
	A, r [Note 3]	2	1	-	A, CY <- A + r + CY	x	x	x
	r, A	2	1	-	r, CY <- r + A + CY	x	x	x
	A, saddr	2	1	-	A, CY <- A + (saddr) + CY	x	x	x
	A, !addr16	3	1	4	A, CY <- A + (addr16) + CY	x	x	x
	A, [HL]	1	1	4	A, CY <- A + (HL) + CY	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A + (HL + byte) + CY	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A + (HL + B) + CY	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A + (HL + C) + CY	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A + (ES, addr16) + CY	x	x	x
	A, ES:[HL]	2	2	5	A, CY <- A + (ES, HL) + CY	x	x	x
	A, ES:[HL+byte]	3	2	5	A, CY <- A + ((ES, HL) + byte) + CY	x	x	x
	A, ES:[HL+B]	3	2	5	A, CY <- A + ((ES, HL) + B) + CY	x	x	x
	A, ES:[HL+C]	3	2	5	A, CY <- A + ((ES, HL) + C) + CY	x	x	x
SUB	A, #byte	2	1	-	A, CY <- A - byte	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) - byte	x	x	x
	A, r [Note 3]	2	1	-	A, CY <- A - r	x	x	x

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
	r, A	2	1	-	r, CY <- r - A	x	x	x
	A, saddr	2	1	-	A, CY <- A - (saddr)	x	x	x
	A, !addr16	3	1	4	A, CY <- A - (addr16)	x	x	x
	A, [HL]	1	1	4	A, CY <- A - (HL)	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A - (HL + byte)	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A - (HL + B)	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A - (HL + C)	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A - (ES:addr16)	x	x	x
	A, ES:[HL]	2	2	5	A, CY <- A - (ES:HL)	x	x	x
	A, ES:[HL+byte]	3	2	5	A, CY <- A - ((ES:HL) + byte)	x	x	x
	A, ES:[HL+B]	3	2	5	A, CY <- A - ((ES:HL) + B)	x	x	x
	A, ES:[HL+C]	3	2	5	A, CY <- A - ((ES:HL) + C)	x	x	x
SUBC	A, #byte	2	1	-	A, CY <- A - byte - CY	x	x	x
	saddr, #byte	3	2	-	(saddr), CY <- (saddr) - byte - CY	x	x	x
	A, r [Note 3]	2	1	-	A, CY <- A - r - CY	x	x	x
	r, A	2	1	-	r, CY <- r - A - CY	x	x	x
	A, saddr	2	1	-	A, CY <- A - (saddr) - CY	x	x	x
	A, !addr16	3	1	4	A, CY <- A - (addr16) - CY	x	x	x
	A, [HL]	1	1	4	A, CY <- A - (HL) - CY	x	x	x
	A, [HL+byte]	2	1	4	A, CY <- A - (HL + byte) - CY	x	x	x
	A, [HL+B]	2	1	4	A, CY <- A - (HL + B) - CY	x	x	x
	A, [HL+C]	2	1	4	A, CY <- A - (HL + C) - CY	x	x	x
	A, ES:!addr16	4	2	5	A, CY <- A - (ES:addr16) - CY	x	x	x
	A, ES:[HL]	2	2	5	A, CY <- A - (ES:HL) - CY	x	x	x
	A, ES:[HL+byte]	3	2	5	A, CY <- A - ((ES:HL) + byte) - CY	x	x	x
	A, ES:[HL+B]	3	2	5	A, CY <- A - ((ES:HL) + B) - CY	x	x	x
	A, ES:[HL+C]	3	2	5	A, CY <- A - ((ES:HL) + C) - CY	x	x	x
AND	A, #byte	2	1	-	A <- A ^ byte	x		
	saddr, #byte	3	2	-	(saddr) <- (saddr) ^ byte	x		
	A, r [Note 3]	2	1	-	A <- A ^ r	x		
	r, A	2	1	-	r <- r ^ A	x		
	A, saddr	2	1	-	A <- A ^ (saddr)	x		
	A, !addr16	3	1	4	A <- A ^ (addr16)	x		
	A, [HL]	1	1	4	A <- A ^ (HL)	x		
	A, [HL+byte]	2	1	4	A <- A ^ (HL + byte)	x		
	A, [HL+B]	2	1	4	A <- A ^ (HL + B)	x		

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
	A, [HL+C]	2	1	4	A <- A ^ (HL + C)	x		
	A, ES:!addr16	4	2	5	A <- A ^ (ES:addr16)	x		
	A, ES:[HL]	2	2	5	A <- A ^ (ES:HL)	x		
	A, ES:[HL+byte]	3	2	5	A <- A ^ ((ES:HL) + byte)	x		
	A, ES:[HL+B]	3	2	5	A <- A ^ ((ES:HL) + B)	x		
	A, ES:[HL+C]	3	2	5	A <- A ^ ((ES:HL) + C)	x		
OR	A, #byte	2	1	-	A <- A v byte	x		
	saddr, #byte	3	2	-	(saddr) <- (saddr) v byte	x		
	A, r [Note 3]	2	1	-	A <- A v r	x		
	r, A	2	1	-	r <- r v A	x		
	A, saddr	2	1	-	A <- A v (saddr)	x		
	A, !addr16	3	1	4	A <- A v (addr16)	x		
	A, [HL]	1	1	4	A <- A v (HL)	x		
	A, [HL+byte]	2	1	4	A <- A v (HL + byte)	x		
	A, [HL+B]	2	1	4	A <- A v (HL + B)	x		
	A, [HL+C]	2	1	4	A <- A v (HL + C)	x		
	A, ES:!addr16	4	2	5	A <- A v (ES:addr16)	x		
	A, ES:[HL]	2	2	5	A <- A v (ES:HL)	x		
	A, ES:[HL+byte]	3	2	5	A <- A v ((ES:HL) + byte)	x		
	A, ES:[HL+B]	3	2	5	A <- A v ((ES:HL) + B)	x		
	A, ES:[HL+C]	3	2	5	A <- A v ((ES:HL) + C)	x		
XOR	A, #byte	2	1	-	A <- A ∀ byte	x		
	saddr, #byte	3	2	-	(saddr) <- (saddr) ∀ byte	x		
	A, r [Note 3]	2	1	-	A <- A ∀ r	x		
	r, A	2	1	-	r <- r ∀ A	x		
	A, saddr	2	1	-	A <- A ∀ (saddr)	x		
	A, !addr16	3	1	4	A <- A ∀ (addr16)	x		
	A, [HL]	1	1	4	A <- A ∀ (HL)	x		
	A, [HL+byte]	2	1	4	A <- A ∀ (HL + byte)	x		
	A, [HL+B]	2	1	4	A <- A ∀ (HL + B)	x		
	A, [HL+C]	2	1	4	A <- A ∀ (HL + C)	x		
	A, ES:!addr16	4	2	5	A <- A ∀ (ES:addr16)	x		
	A, ES:[HL]	2	2	5	A <- A ∀ (ES:HL)	x		
	A, ES:[HL+byte]	3	2	5	A <- A ∀ ((ES:HL) + byte)	x		
	A, ES:[HL+B]	3	2	5	A <- A ∀ ((ES:HL) + B)	x		
	A, ES:[HL+C]	3	2	5	A <- A ∀ ((ES:HL) + C)	x		

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
CMP	A, #byte	2	1	-	A - byte	x	x	x
	saddr, #byte	3	1	-	(saddr) - byte	x	x	x
	A, r [Note 3]	2	1	-	A - r	x	x	x
	r, A	2	1	-	r - A	x	x	x
	A, saddr	2	1	-	A - (saddr)	x	x	x
	A, !addr16	3	1	4	A - (addr16)	x	x	x
	A, [HL]	1	1	4	A - (HL)	x	x	x
	A, [HL+byte]	2	1	4	A - (HL + byte)	x	x	x
	A, [HL+B]	2	1	4	A - (HL + B)	x	x	x
	A, [HL+C]	2	1	4	A - (HL + C)	x	x	x
	!addr16, #byte	4	1	4	(addr16) - byte	x	x	x
	A, ES:!addr16	4	2	5	A - (ES:addr16)	x	x	x
	A, ES:[HL]	2	2	5	A - (ES:HL)	x	x	x
	A, ES:[HL+byte]	3	2	5	A - ((ES:HL) + byte)	x	x	x
	A, ES:[HL+B]	3	2	5	A - ((ES:HL) + B)	x	x	x
	A, ES:[HL+C]	3	2	5	A - ((ES:HL) + C)	x	x	x
	ES:!addr16, #byte	5	2	5	(ES:addr16) - byte	x	x	x
CMP0	A	1	1	-	A - 00H	x	x	x
	X	1	1	-	X - 00H	x	x	x
	B	1	1	-	B - 00H	x	x	x
	C	1	1	-	C - 00H	x	x	x
	saddr	2	1	-	(saddr) - 00H	x	x	x
	!addr16	3	1	4	(addr16) - 00H	x	x	x
	ES:!addr16	4	2	5	(ES:addr16) - 00H	x	x	x
CMPS	X, [HL+byte]	3	1	4	X - (HL + byte)	x	x	x
	X, ES:[HL+byte]	4	2	5	X - ((ES:HL) + byte)	x	x	x

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. Except r = A.

4. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to

the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(d) 16-bit operation instructions

Table 4-34. Operation List (16-bit Operation Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
ADDW	AX, #word	3	1	-	AX, CY <- AX + word	x	x	x
	AX, AX	1	1	-	AX, CY <- AX + AX	x	x	x
	AX, BC	1	1	-	AX, CY <- AX + BC	x	x	x
	AX, DE	1	1	-	AX, CY <- AX + DE	x	x	x
	AX, HL	1	1	-	AX, CY <- AX + HL	x	x	x
	AX, saddrp	2	1	-	AX, CY <- AX + (saddrp)	x	x	x
	AX, !addr16	3	1	4	AX, CY <- AX + (addr16)	x	x	x
	AX, [HL+byte]	3	1	4	AX, CY <- AX + (HL + byte)	x	x	x
	AX, ES:!addr16	4	2	5	AX, CY <- AX + (ES:addr16)	x	x	x
	AX, ES:[HL+byte]	4	2	5	AX, CY <- AX + ((ES:HL) + byte)	x	x	x
SUBW	AX, #word	3	1	-	AX, CY <- AX - word	x	x	x
	AX, BC	1	1	-	AX, CY <- AX - BC	x	x	x
	AX, DE	1	1	-	AX, CY <- AX - DE	x	x	x
	AX, HL	1	1	-	AX, CY <- AX - HL	x	x	x
	AX, saddrp	2	1	-	AX, CY <- AX - (saddrp)	x	x	x
	AX, !addr16	3	1	4	AX, CY <- AX - (addr16)	x	x	x
	AX, [HL+byte]	3	1	4	AX, CY <- AX - (HL - byte)	x	x	x
	AX, ES:!addr16	4	2	5	AX, CY <- AX - (ES:addr16)	x	x	x
	AX, ES:[HL+byte]	4	2	5	AX, CY <- AX - ((ES:HL) + byte)	x	x	x
CMPW	AX, #word	3	1	-	AX - word	x	x	x
	AX, BC	1	1	-	AX - BC	x	x	x
	AX, DE	1	1	-	AX - DE	x	x	x
	AX, HL	1	1	-	AX - HL	x	x	x
	AX, saddrp	2	1	-	AX - (saddrp)	x	x	x
	AX, !addr16	3	1	4	AX - (addr16)	x	x	x
	AX, [HL+byte]	3	1	4	AX - (HL + byte)	x	x	x
	AX, ES:!addr16	4	2	5	AX - (ES:addr16)	x	x	x
	AX, ES:[HL+byte]	4	2	5	AX - ((ES:HL) + byte)	x	x	x

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result

R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(e) Multiply instructions

Table 4-35. Operation List (Multiply Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
MULU	X	1	1	-	AX <- A x X			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(f) Increase-decrease instructions

Table 4-36. Operation List (Increase-Decrease Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
INC	r	1	1	-	r <- r + 1	x	x	
	saddr	2	2	-	(saddr) <- (saddr) + 1	x	x	
	!addr16	3	2	-	(addr16) <- (addr16) + 1	x	x	
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) + 1	x	x	
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) + 1	x	x	
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) + 1	x	x	
DEC	r	1	1	-	r <- r - 1	x	x	
	saddr	2	2	-	(saddr) <- (saddr) - 1	x	x	
	!addr16	3	2	-	(addr16) <- (addr16) - 1	x	x	
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) - 1	x	x	
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) - 1	x	x	
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) - 1	x	x	
INCW	rp	1	1	-	rp <- rp + 1			
	saddrp	2	2	-	(saddrp) <- (saddrp) + 1			
	!addr16	3	2	-	(addr16) <- (addr16) + 1			
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) + 1			
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) + 1			
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) + 1			
DECW	rp	1	1	-	rp <- rp - 1			
	saddrp	2	2	-	(saddrp) <- (saddrp) - 1			
	!addr16	3	2	-	(addr16) <- (addr16) - 1			
	[HL+byte]	3	2	-	(HL + byte) <- (HL + byte) - 1			
	ES:!addr16	4	3	-	(ES, addr16) <- (ES, addr16) - 1			
	ES: [HL+byte]	4	3	-	((ES:HL) + byte) <- ((ES:HL) + byte) - 1			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result

R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(g) Shift instructions

Table 4-37. Operation List (Shift Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
SHR	A, cnt	2	1	-	(CY <- A_0, A_{m-1} <- A_m, A_7 <- 0) x cnt			x
SHRW	AX, cnt	2	1	-	(CY <- AX_0, AX_{m-1} <- AX_m, AX_{15} <- 0) x cnt			x
SHL	A, cnt	2	1	-	(CY <- A_7, A_m <- A_{m-1}, A_0 <- 0) x cnt			x
	B, cnt	2	1	-	(CY <- B_7, B_m <- B_{m-1}, B_0 <- 0) x cnt			x
	C, cnt	2	1	-	(CY <- C_7, C_m <- C_{m-1}, C_0 <- 0) x cnt			x
SHLW	AX, cnt	2	1	-	(CY <- AX_{15}, AX_m <- AX_{m-1}, AX_0 <- 0) x cnt			x
	BC, cnt	2	1	-	(CY <- BC_{15}, BC_m <- BC_{m-1}, BC_0 <- 0) x cnt			x
SAR	A, cnt	2	1	-	(CY <- A_0, A_{m-1} <- A_m, A_7 <- A_7) xcnt			x
SARW	AX, cnt	2	1	-	(CY <- AX_0, AX_{m-1} <- AX_m, AX_{15} <- AX_{15}) x cnt			x

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. cnt is the number of bit shifts.

4. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(h) Rrotate instructions

Table 4-38. Operation List (Rotate Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
ROR	A, 1	2	1	-	(CY, A_7 <- A_0, A_{m-1} <- A_m)x1			x
ROL	A, 1	2	1	-	(CY, A_0 <- A_7, A_{m+1} <- A_m)x1			x
RORC	A, 1	2	1	-	(CY <- A_0, A_7 <- CY, A_{m-1} <- A_m)x1			x
ROLC	A, 1	2	1	-	(CY <- A_7, A_0 <- CY, A_{m+1} <- A_m)x1			x
ROLWC	AX, 1	2	1	-	(CY <- AX_{15}, AX_0 <- CY, AX_{m+1} <- AX_m)x1			x
	BC, 1	2	1	-	(CY <- BC_{15}, BC_0 <- CY, BC_{m+1} <- BC_m)x1			x

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(i) Bit manipulation instructions

Table 4-39. Operation List (Bit Manipulation Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
MOV1	CY, saddr.bit	3	1	-	CY <- (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- sfr.bit			x
	CY, A.bit	2	1	-	CY <- A.bit			x
	CY, PSW.bit	3	1	-	CY <- PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- (HL).bit			x
	saddr.bit, CY	3	2	-	(saddr).bit <- CY			
	sfr.bit, CY	3	2	-	sfr.bit <- CY			
	A.bit, CY	2	1	-	A.bit <- CY			
	PSW.bit, CY	3	4	-	PSW.bit <- CY	x	x	
	[HL].bit, CY	2	2	-	(HL).bit <- CY			
	CY, ES:[HL].bit	3	2	5	CY <- (ES, HL).bit			x
	ES:[HL].bit, CY	3	3	-	(ES, HL).bit <- CY			
AND1	CY, saddr.bit	3	1	-	CY <- CY ^ (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- CY ^ sfr.bit			x
	CY, A.bit	2	1	-	CY <- CY ^ A.bit			x
	CY, PSW.bit	3	1	-	CY <- CY ^ PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- CY ^ (HL).bit			x
	CY, ES:[HL].bit	3	2	5	CY <- CY ^ (ES, HL).bit			x
OR1	CY, saddr.bit	3	1	-	CY <- CY v (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- CY v sfr.bit			x
	CY, A.bit	2	1	-	CY <- CY v A.bit			x
	CY, PSW.bit	3	1	-	CY <- CY v PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- CY v (HL).bit			x
	CY, ES:[HL].bit	3	2	5	CY <- CY v (ES, HL).bit			x
XOR1	CY, saddr.bit	3	1	-	CY <- CY ∀ (saddr).bit			x
	CY, sfr.bit	3	1	-	CY <- CY ∀ sfr.bit			x
	CY, A.bit	2	1	-	CY <- CY ∀ A.bit			x
	CY, PSW.bit	3	1	-	CY <- CY ∀ PSW.bit			x
	CY, [HL].bit	2	1	4	CY <- CY ∀ (HL).bit			x
	CY, ES:[HL].bit	3	2	5	CY <- CY ∀ (ES, HL).bit			x
SET1	saddr.bit	3	2	-	(saddr).bit <- 1			
	sfr.bit	3	2	-	sfr.bit <- 1			
	A.bit	2	1	-	A.bit <- 1			
	!addr16.bit	4	2	-	(addr16).bit <- 1			
	PSW.bit	3	4	-	PSW.bit <- 1	x	x	x
	[HL].bit	2	2	-	(HL).bit <- 1			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
	ES:!addr16.bit	5	3	-	(ES, addr16).bit <- 1			
	ES:[HL].bit	3	3	-	(ES, HL).bit <- 1			
CLR1	saddr.bit	3	2	-	(saddr).bit <- 0			
	sfr.bit	3	2	-	sfr.bit <- 0			
	A.bit	2	1	-	A.bit <- 0			
	!addr16.bit	4	2	-	(addr16).bit <- 0			
	PSW.bit	3	4	-	PSW.bit <- 0	x	x	x
	[HL].bit	2	2	-	(HL).bit <- 0			
	ES:!addr16.bit	5	3	-	(ES, addr16).bit <- 0			
	ES:[HL].bit	3	3	-	(ES, HL).bit <- 0			
SET1	CY	2	1	-	CY <- 1			1
CLR1	CY	2	1	-	CY <- 0			0
NOT1	CY	2	1	-	CY <- $\overline{CY}$			x

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(j) Call return instructions

Table 4-40. Operations List (Call Return Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
CALL	rp	2	3	-	(SP - 2) <- (PC + 2)S, (SP - 3) <- (PC + 2)H, (SP - 4) <- (PC + 2)L, PC <- CS, rp, SP <- SP - 4			
	$!addr20	3	3	-	(SP - 2) <- (PC + 3)S, (SP - 3) <- (PC + 3)H, (SP - 4) <- (PC + 3)L, PC <- PC + 3 + jdisp16, SP <- SP - 4			
	!addr16	3	3	-	(SP - 2) <- (PC + 3)S, (SP - 3) <- (PC + 3)H, (SP - 4) <- (PC + 3)L, PC <- 0000, addr16, SP <- SP - 4			
	!!addr20	4	3	-	(SP - 2) <- (PC + 4)S, (SP - 3) <- (PC + 4)H, (SP - 4) <- (PC + 4)L, PC <- addr20, SP <- SP - 4			
CALLT	[addr5]	2	5	-	(SP - 2) <- (PC + 2)S, (SP - 3) <- (PC + 2)H, (SP - 4) <- (PC + 2)L, PCS <- 0000, PCH <- (000000000000, addr5 + 1), PCL <- (000000000000, addr5), SP <- SP - 4			
BRK	-	2	5	-	(SP - 1) <- PSW, (SP - 2) <- (PC + 2)S, (SP - 3) <- (PC + 2)H, (SP - 4) <- (PC + 2)L, PCS <- 0000, PCH <- (0007FH), PCL <- (0007EH), SP <- SP - 4, IE <- 0			
RET	-	1	6	-	PCL <- (SP), PCH <- (SP + 1), PCS <- (SP + 2), SP <- SP + 4			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
RETI	-	2	6	-	PC_L <- (SP), PC_H <- (SP + 1), PC_S <- (SP + 2), PSW <- (SP + 3), SP <- SP + 4	R	R	R
RETB	-	2	6	-	PC_L <- (SP), PC_H <- (SP + 1), PC_S <- (SP + 2), PSW <- (SP + 3), SP <- SP + 4	R	R	R

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(k) Stack manipulation instructions

Table 4-41. Operation List (Stack Manipulation Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
PUSH	PSW	2	1	-	(SP - 1) <- PSW, (SP - 2) <- 00H, SP <- SP - 2			
	rp	1	1	-	(SP - 1) <- rp_H, (SP - 2) <- rp_L, SP <- SP - 2			
POP	PSW	2	3	-	PSW <- (SP + 1), SP <- SP + 2	R	R	R
	rp	1	1	-	rp_L <- (SP), rp_H <- (SP + 1), SP <- SP + 2			
MOVW	SP, #word	4	1	-	SP <- word			
	SP, AX	2	1	-	SP <- AX			
	AX, SP	2	1	-	AX <- SP			
	HL, SP	3	1	-	HL <- SP			
	BC, SP	3	1	-	BC <- SP			
	DE, SP	3	1	-	DE <- SP			
ADDW	SP, #byte	2	1	-	SP <- SP + byte			
SUBW	SP, #byte	2	1	-	SP <- SP - byte			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(l) Unconditional branch instructions

Table 4-42. Operation List (Unconditional Branch Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
BR	AX	2	3	-	PC <- CS, AX			
	$addr20	2	3	-	PC <- PC + 2 + jdisp8			
	$!addr20	3	3	-	PC <- PC + 3 + jdisp16			
	!addr16	3	3	-	PC <- 0000, addr16			
	!!addr20	4	3	-	PC <- addr20			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(m) Conditional branch instructions

Table 4-43. Operation List (Conditional Branch Instructions)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
BC	$addr20	2	2/4[Note 3]	-	PC <- PC + 2 + jdisp8 if CY = 1			
BNC	$addr20	2	2/4[Note 3]	-	PC <- PC + 2 + jdisp8 if CY = 0			
BZ	$addr20	2	2/4[Note 3]	-	PC <- PC + 2 + jdisp8 if Z = 1			
BNZ	$addr20	2	2/4[Note 3]	-	PC <- PC + 2 + jdisp8 if Z = 0			
BH	$addr20	3	2/4[Note 3]	-	PC <- PC + 3 + jdisp8 if (Z v CY) = 0			
BNH	$addr20	3	2/4[Note 3]	-	PC <- PC + 3 + jdisp8 if (Z v CY) = 1			
BT	saddr.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if (saddr).bit = 1			
	sfr.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if sfr.bit = 1			
	A.bit, $addr20	3	3/5[Note 3]	-	PC <- PC + 3 + jdisp8 if A.bit = 1			
	PSW.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if PSW.bit = 1			
	[HL].bit, $addr20	3	3/5[Note 3]	6/8	PC <- PC + 3 + jdisp8 if (HL).bit = 1			
	ES:[HL].bit, $addr20	4	4/6[Note 3]	7/9	PC <- PC + 4 + jdisp8 if (ES, HL).bit = 1			
BF	saddr.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if (saddr).bit = 0			
	sfr.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if sfr.bit = 0			
	A.bit, $addr20	3	3/5[Note 3]	-	PC <- PC + 3 + jdisp8 if A.bit = 0			
	PSW.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if PSW.bit = 0			
	[HL].bit, $addr20	3	3/5[Note 3]	6/8	PC <- PC + 3 + jdisp8 if (HL).bit = 0			
	ES:[HL].bit, $addr20	4	4/6[Note 3]	7/9	PC <- PC + 4 + jdisp8 if (ES, HL).bit = 0			
BTCLR	saddr.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if (saddr).bit = 1 then reset (saddr).bit			
	sfr.bit, $addr20	4	3/5[Note 3]	-	PC <- PC + 4 + jdisp8 if sfr.bit = 1 then reset sfr.bit			
	A.bit, $addr20	3	3/5[Note 3]	-	PC <- PC + 3 + jdisp8 if A.bit = 1 then reset A.bit			

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 4]		
			Note 1	Note 2		Z	AC	CY
	PSW.bit, $addr20	4	5/7[Note 3]	-	PC <- PC + 4 + jdisp8 if PSW.bit = 1 then reset PSW.bit	x	x	x
	[HL].bit, $addr20	3	3/5[Note 3]	-	PC <- PC + 3 + jdisp8 if (HL).bit = 1 then reset (HL).bit			
	ES:[HL].bit, $addr20	4	4/6[Note 3]	-	PC <- PC + 4 + jdisp8 if (ES, HL).bit = 1 then reset (ES, HL).bit			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The clock number shows the condition satisfied or condition unsatisfied.

4. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(n) Conditional step instruction

Table 4-44. Operation List (Conditional Step Instruction)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
SKC	-	2	1	-	Next instruction skip if CY = 1			
SKNC	-	2	1	-	Next instruction skip if CY = 0			
SKZ	-	2	1	-	Next instruction skip if Z = 1			
SKNZ	-	2	1	-	Next instruction skip if Z = 0			
SKH	-	2	1	-	Next instruction skip if (Z v CY) = 0			
SKNH	-	2	1	-	Next instruction skip if (Z v CY) = 1			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

(o) CPU control instruction

Table 4-45. Operation List (CPU Control Instruction)

Mnemonic	Operand	Byte	Clock		Operation	Flag[Note 3]		
			Note 1	Note 2		Z	AC	CY
SEL	RBn	2	1	-	RBS[1:0] <- n			
NOP	-	1	1	-	No Operation			
EI	-	3	4	-	IE <- 1 (Enable Interrupt)			
DI	-	3	4	-	IE <- 0 (Disable Interrupt)			
HALT	-	2	3	-	Set HALT Mode			
STOP	-	2	3	-	Set STOP Mode			

Notes 1. When internal RAM area or SFR area is accessed or using the instruction for no data access.

2. When program memory area is accessed.

3. The flag field symbol shows the flag change at the time the instruction is executed.

Blank : Unchanged
0 : Cleared to 0
1 : Set to 1
x : Set or cleared according to the result
R : Previously saved value is restored

Remarks 1. One clock of the instruction is one clock of the CPU clock (fCLK) that is selected by the processor clock control register (PCC).

2. Clock number when there is a program in the internal ROM (Flash memory) area.

3. n is the number of register banks (n = 0 to 3).

4. In products in which external memory area connects to internal flash area, when using external bus interface function, wait number is added to the instruction execution clock number that is mapped to the final address of Flash (maximum 16 bytes). This is because when carrying out a prior read of the instruction code, flash space is exceeded and external memory space is accessed and so an external memory wait is entered. Please see "(b) Access to external memory contents as data" for wait numbers.

4.6.6 Explanation of instructions

This section explains the instructions of 78K0R microcontrollers.

Table 4-46. Assembly Language Instruction List

Function	Instruction
8-bit data transmission instructions	MOV, XCH, ONEB, CLRB, MOVS
16-bit data transmission instructions	MOVW, XCHW, ONEW, CLRW
8-bit operation instructions	ADD, ADDC, SUB, SUBC, AND, OR, XOR, CMP, CMP0, CMPS
16-bit operation instructions	ADDW, SUBW, CMPW
Multiply instruction	MULU
Increment/decrement instructions	INC, DEC, INCW, DECW
Shift instructions	SHR, SHRW, SHL, SHLW, SAR, SARW
Rotate instructions	ROR, ROL, RORC, ROLC, ROLWC
Bit manipulation instructions	MOV1, AND1, OR1, XOR1, SET1, CLR1, NOT1
Call return instructions	CALL, CALLT, BRK, RET, RETI, RETB
Stack manipulation instructions	PUSH, POP, MOVW, ADDW, SUBW
Unconditional branch instruction	BR
Conditional branch instructions	BC, BNC, BZ, BNZ, BH, BNH, BT, BF, BTCLR
Conditional skip instructions	SKC, SKNC, SKZ, SKNZ, SKH, SKNH
CPU control instructions	SEL, NOP, EI, DI, HALT, STOP

The following information explains the individual instructions.

[Instruction format]

Shows the basic written format of the instruction.

[Operation]

The instruction operation is shown by using the code address.

[Operand]

The operand that can be specified with this instruction is shown. Please see "(2) Operation field symbols" for descriptions of each operand.

[Flag]

Indicates the flag operation that changes by instruction execution.
Each flag operation symbol is shown in the conventions.

Symbol	Description
Blank	Unchanged
0	Cleared to 0
1	Set to 1
x	Set or cleared according to the result
R	Previously saved value is restored

[Description]

Describes the instruction operation in detail.

[Description example]

Description example of an instruction is indicated.

(1) 8-bit data transmission instructions

The following 8-bit data transmission instructions are available.

Instruction	Overview
MOV	Byte data transfer
XCH	Byte data exchange
ONEB	Byte data 01H set
CLRB	Byte data clear
MOVS	Byte data transfer and PSW change

MOV

Byte data transfer

[Instruction format]

MOV dst, src

[Operation]

dst <- src

[Operand]

Operand (dst, src)
r, #byte
saddr, #byte
sfr, #byte
!addr16, #byte
A, r[Note]
r, A[Note]
A, saddr
saddr, A
A, sfr
sfr, A
A, !addr16
!addr16, A
PSW, #byte
A, PSW
PSW, A
ES, #byte
ES, saddr
A, ES
ES, A
CS, #byte
A, CS
CS, A
A, [DE]
[DE], A
[DE+byte], #byte
A, [DE+byte]
[DE+byte], A
A, [HL]
[HL], A

Operand (dst, src)
[HL+byte], #byte
A, [HL+byte]
[HL+byte], A
A, [HL+B]
[HL+B], A
A, [HL+C]
[HL+C], A
word[B], #byte
A, word[B]
word[B], A
word[C], #byte
A, word[C]
word[C], A
word[BC], #byte
A, word[BC]
word[BC], A
[SP+byte], #byte
A, [SP+byte]
[SP+byte], A
B, saddr
B, !addr16
C, saddr
C, !addr16
X, saddr
X, !addr16
ES:!addr16, #byte
A, ES:!addr16
ES:!addr16, A
A, ES:[DE]
ES:[DE], A
ES:[DE+byte], #byte
A, ES:[DE+byte]
ES:[DE+byte], A
A, ES:[HL]
ES:[HL], A
ES:[HL+byte], #byte
A, ES:[HL+byte]
ES:[HL+byte], A
A, ES:[HL+B]

Operand (dst, src)
ES:[HL+B], A
A, ES:[HL+C]
ES:[HL+C], A
ES:word[B], #byte
A, ES:word[B]
ES:word[B], A
ES:word[C], #byte
A, ES:word[C]
ES:word[C], A
ES:word[BC], #byte
A, ES:word[BC]
ES:word[BC], A
B, ES:!addr16
C, ES:!addr16
X, ES:!addr16

Note Except r = A.

[Flag]

(1) PSW, #byte and PSW, A operands

Z	AC	CY
x	x	x

x : Set or cleared according to the result

(2) All other operand combinations

Z	AC	CY

Blank : Unchanged

[Description]

- The contents of the source operand (src) specified by the 2nd operand are transferred to the destination operand (dst) specified by the 1st operand.
- No interrupts are acknowledged between the MOV PSW, #byte instruction/MOV PSW, A instruction and the next instruction.

[Description example]

```
MOV     A, #4DH         ;  (1)
```

(1) 4DH is transferred to the A register.

XCH

Byte data exchange

[Instruction format]

XCH dst, src

[Operation]

dst <--> src

[Operand]

Operand (dst, src)
A, r[Note]
A, saddr
A, sfr
A, !addr16
A, [DE]
A, [DE+byte]
A, [HL]
A, [HL + byte]
A, [HL + B]
A, [HL + C]
A, ES:!addr16
A, ES:[DE]
A, ES:[DE+byte]
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The 1st and 2nd operand contents are exchanged.

[Description example]

```
XCH     A, FFEBCH       ; (1)
```

(1) The A register contents and address FFEBCH contents are exchanged.

ONEB

Byte data 01H set

[Instruction format]

ONEB dst

[Operation]

dst <- 01H

[Operand]

Operand (dst)
A
X
B
C
saddr
!addr16
ES:!addr16

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- 01H is transferred to the destination operand (dst) specified by the first operand.

[Description example]

```
ONEB    A       ; (1)
```

(1) Transfers 01H to the A register.

CLRB

Byte data clear

[Instruction format]

CLRB dst

[Operation]

dst <- 00H

[Operand]

Operand (dst)
A
X
B
C
saddr
!addr16
ES:!addr16

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- 00H is transferred to the destination operand (dst) specified by the first operand.

[Description example]

```
CLRB    A       ; (1)
```

(1) Transfers 00H to the A register.

MOVS

Byte data transfer and PSW change

[Instruction format]

MOVS dst, src

[Operation]

dst <- src

[Operand]

Operand (dst, src)
[HL+byte], X
ES:[HL+byte], X

[Flag]

Z	AC	CY
x		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The contents of the source operand specified by the second operand is transferred to the destination operand (dst) specified by the first operand.
- If the src value is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the register A value is 0 or if the src value is 0, the CY flag is set (1). In all other cases, the CY flag is cleared (0).

[Description example]

```
MOVS    [HL+2H], X      ; (1)
```

(1) When HL = FE00H, X = 55H, A = 0H : "X = 55H" is stored at address FE02H.
Z flag = 0 CY flag = 1 (since A register = 0)

(2) 16-bit data transmission instructions

The following 16-bit data transmission instructions are available.

Instruction	Overview
MOVW	Word data transfer
XCHW	Word data exchange
ONEW	Word data 0001H set
CLRW	Word data clear

MOVW

Word data transfer

[Instruction format]

MOVW dst, src

[Operation]

dst <- src

[Operand]

Operand (dst, src)
rp, #word
saddrp, #word
sfrp, #word
AX, saddrp
saddrp, AX
AX, sfrp
sfrp, AX
AX, rp[Note]
rp, AX[Note]
AX, !addr16
!addr16, AX
AX, [DE]
[DE], AX
AX, [DE+byte]
[DE+byte], AX
AX, [HL]
[HL], AX
AX, [HL+byte]
[HL+byte], AX
AX, word[B]
word[B], AX
AX, word[C]
word[C], AX
AX, word[BC]
word[BC], AX
AX, [SP+byte]
[SP+byte], AX
BC, saddrp
BC, !addr16

Operand (dst, src)
DE, saddrp
DE, !addr16
HL, saddrp
HL, !addr16
AX, ES:!addr16
ES:!addr16, AX
AX, ES:[DE]
ES:[DE], AX
AX, ES:[DE+byte]
ES:[DE+byte], AX
AX, ES:[HL]
ES:[HL], AX
AX, ES:[HL+byte]
ES:[HL+byte], AX
AX, ES:word[B]
ES:word[B], AX
AX, ES:word[C]
ES:word[C], AX
AX, ES:word[BC]
ES:word[BC], AX
BC, ES:!addr16
DE, ES:!addr16
HL, ES:!addr16

Note Only when rp = BC, DE or HL

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The contents of the source operand (src) specified by the 2nd operand are transferred to the destination operand (dst) specified by the 1st operand.

[Description example]

```
MOVW    AX, HL          ;  (1)
```

(1) The HL register contents are transferred to the AX register.

[Cautions]

- Only an even address can be specified. An odd address cannot be specified.

XCHW

Word data exchange

[Instruction format]

XCHW dst, src

[Operation]

dst <--> src

[Operand]

Operand (dst, src)
AX, rp[Note]

Note Only when rp = BC, DE or HL

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The 1st and 2nd operand contents are exchanged.

[Description example]

```
XCHW    AX, BC          ;  (1)
```

(1) The memory contents of the AX register are exchanged with those of the BC register.

ONEW

Word data 0001H set

[Instruction format]

ONEW dst

[Operation]

dst <- 0001H

[Operand]

Operand (dst)
AX
BC

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- 0001H is transferred to the destination operand (dst) specified by the first operand.

[Description example]

```
ONEW    AX      ; (1)
```

(1) 0001H is transferred to the AX register.

CLRW

Word data clear

[Instruction format]

CLRW dst

[Operation]

dst <- 0000H

[Operand]

Operand (dst)
AX
BC

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- 0000H is transferred to the destination operand (dst) specified by the first operand.

[Description example]

```
CLRW    AX      ; (1)
```

(1) 0000H is transferred to the AX register.

(3) 8-bit operation instructions

The following 8-bit operation instructions are available.

Instruction	Overview
ADD	Byte data addition
ADDC	Byte data addition including carry
SUB	Byte data subtraction
SUBC	Byte data subtraction including carry
AND	Byte data AND operation
OR	Byte data OR operation
XOR	Byte data exclusive OR operation
CMP	Byte data comparison
CMP0	Byte data zero comparison
CMPS	Byte data comparison

ADD

Byte data addition

[Instruction format]

ADD dst, src

[Operation]

dst, CY <- dst + src

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the 1st operand is added to the source operand (src) specified by the 2nd operand and the result is stored in the CY flag and the destination operand (dst).
- If the addition result shows that dst is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the addition generates a carry out of bit 7, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- If the addition generates a carry for bit 4 out of bit 3, the AC flag is set (1). In all other cases, the AC flag is cleared (0).

[Description example]

```
ADD     CR10, #56H      ;  (1)
```

(1) 56H is added to the CR10 register and the result is stored in the CR10 register.

ADDC

Byte data addition including carry

[Instruction format]

ADDC dst, src

[Operation]

dst, CY <- dst + src + CY

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the 1st operand, the source operand (src) specified by the 2nd operand and the CY flag are added and the result is stored in the destination operand (dst) and the CY flag. The CY flag is added to the least significant bit. This instruction is mainly used to add two or more bytes.
- If the addition result shows that dst is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the addition generates a carry out of bit 7, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- If the addition generates a carry for bit 4 out of bit 3, the AC flag is set (1). In all other cases, the AC flag is cleared (0).

[Description example]

```
ADDC    A, [HL+B]       ;  (1)
```

(1) The A register contents and the contents at address (HL register + (B register)) and the CY flag are added and the result is stored in the A register.

SUB

Byte data subtraction

[Instruction format]

SUB dst, src

[Operation]

dst, CY <- dst - src

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The source operand (src) specified by the 2nd operand is subtracted from the destination operand (dst) specified by the 1st operand and the result is stored in the destination operand (dst) and the CY flag.
  The destination operand can be cleared to 0 by equalizing the source operand (src) and the destination operand (dst).
- If the subtraction shows that dst is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the subtraction generates a borrow out of bit 7, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- If the subtraction generates a borrow for bit 3 out of bit 4, the AC flag is set (1). In all other cases, the AC flag is cleared (0).

[Description example]

```
SUB     D, A            ;  (1)
```

(1) The A register is subtracted from the D register and the result is stored in the D register.

SUBC

Byte data subtraction including carry

[Instruction format]

SUBC dst, src

[Operation]

dst, CY <- dst - src - CY

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The source operand (src) specified by the 2nd operand and the CY flag are subtracted from the destination operand (dst) specified by the 1st operand and the result is stored in the destination operand (dst).
  The CY flag is subtracted from the least significant bit. This instruction is mainly used for subtraction of two or more bytes.
- If the subtraction shows that dst is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the subtraction generates a borrow out of bit 7, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- If the subtraction generates a borrow for bit 3 out of bit 4, the AC flag is set (1). In all other cases, the AC flag is cleared (0).

[Description example]

```
SUBC    A, [HL]         ;  (1)
```

(1) The (HL register) address contents and the CY flag are subtracted from the A register and the result is stored in the A register.

AND

Byte data AND operation

[Instruction format]

AND dst, src

[Operation]

dst <- dst ^ src

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x		

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- Bit-wise logical product is obtained from the destination operand (dst) specified by the 1st operand and the source operand (src) specified by the 2nd operand and the result is stored in the destination operand (dst).
- If the logical product shows that all bits are 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).

[Description example]

```
AND     FFEBAH, #11011100B      ; (1)
```

(1) Bit-wise logical product of FFEBAH contents and 11011100B is obtained and the result is stored at FFEBAH.

OR

Byte data OR operation

[Instruction format]

OR dst, src

[Operation]

dst <- dst v src

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x		

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The bit-wise logical sum is obtained from the destination operand (dst) specified by the 1st operand and the source operand (src) specified by the 2nd operand and the result is stored in the destination operand (dst).
- If the logical sum shows that all bits are 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).

[Description example]

```
OR      A, FFE98H       ; (1)
```

(1) The bit-wise logical sum of the A register and FFE98H is obtained and the result is stored in the A register.

XOR

Byte data exclusive OR operation

[Instruction format]

XOR dst, src

[Operation]

dst <- dst ∀ src

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]

Note Except r = A.

[Flag]

Z	AC	CY
x		

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The bit-wise exclusive logical sum is obtained from the destination operand (dst) specified by the 1st operand and the source operand (src) specified by the 2nd operand and the result is stored in the destination operand (dst).
  Logical negation of all bits of the destination operand (dst) is possible by selecting #0FFH for the source operand (src) with this instruction.
- If the exclusive logical sum shows that all bits are 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).

[Description example]

```
XOR     A, L            ;  (1)
```

(1) The bit-wise exclusive logical sum of the A and L registers is obtained and the result is stored in the A register.

CMP

Byte data comparison

[Instruction format]

CMP dst, src

[Operation]

dst - src

[Operand]

Operand (dst, src)
A, #byte
saddr, #byte
A, r[Note]
r, A
A, saddr
A, !addr16
A, [HL]
A, [HL+byte]
A, [HL+B]
A, [HL+C]
!addr16, #byte
A, ES:!addr16
A, ES:[HL]
A, ES:[HL+byte]
A, ES:[HL+B]
A, ES:[HL+C]
ES:!addr16, #byte

Note Except r = A.

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The source operand (src) specified by the 2nd operand is subtracted from the destination operand (dst) specified by the 1st operand.
  The subtraction result is not stored anywhere and only the Z, AC and CY flags are changed.
- If the subtraction result is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the subtraction generates a borrow out of bit 7, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- If the subtraction generates a borrow for bit 3 out of bit 4, the AC flag is set (1). In all other cases, the AC flag is cleared (0).

[Description example]

```
CMP     FFE38H, #38H            ; (1)
```

(1) 38H is subtracted from the contents at address FFE38H and only the flags are changed (comparison of contents at address FFE38H and the immediate data).

CMP0

Byte data zero comparison

[Instruction format]

CMP0 dst

[Operation]

dst - 00H

[Operand]

Operand (dst)
A
X
B
C
saddr
!addr16
ES:!addr16

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- 00H is subtracted from the destination operand (dst) specified by the first operand.
- The subtraction result is not stored anywhere and only the Z, AC and CY flags are changed.
- If the dst value is already 00H, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- The AC and CY flags are always cleared (0).

[Description example]

```
CMP0    A               ; (1)
```

(1) The Z flag is set if the A register value is 0.

CMPS

Byte data comparison

[Instruction format]

CMPS dst, src

[Operation]

dst - src

[Operand]

Operand (dst, src)
X, [HL+byte]
X, ES:[HL+byte]

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The source operand (src) specified by the 2nd operand is subtracted from the destination operand (dst) specified by the 1st operand.
  The subtraction result is not stored anywhere and only the Z, AC and CY flags are changed.
- If the subtraction result is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- When the calculation result is not 0 or when the value of either register A or dst is 0, then the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- If the subtraction generates a borrow out of bit 4 to bit 3, the AC flag is set (1). In all other cases, the AC flag is cleared (0).

[Description example]

```
CMPS    X, [HL+F0H]     ; (1)
```

(1) When HL = FD12H : The value of X is compared with the contents of address FFE02H, and the Z flag is set if the two values match. The value of X is compared with the contents of address FFE02H, and the CY flag is set if the two values do not match.
The CY flag is set when the value of register A is 0. The CY flag is set when the value of register X is 0. The AC flag is set by borrowing from bit 4 to bit 3, similar to the CMP instruction.

(4) 16-bit operation instructions

The following 16-bit operation instructions are available.

Instruction	Overview
ADDW	Word data addition
SUBW	Word data subtraction
CMPW	Word data comparison

ADDW

Word data addition

[Instruction format]

ADDW dst, src

[Operation]

dst, CY <- dst + src

[Operand]

Operand (dst, src)
AX, #word
AX, AX
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the 1st operand is added to the source operand (src) specified by the 2nd operand and the result is stored in the destination operand (dst).
- If the addition result shows that dst is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the addition generates a carry out of bit 15, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- As a result of addition, the AC flag becomes undefined.

[Description example]

```
ADDW    AX, #ABCDH      ;  (1)
```

(1) ABCDH is added to the AX register and the result is stored in the AX register.

SUBW

Word data subtraction

[Instruction format]

SUBW dst, src

[Operation]

dst, CY <- dst - src

[Operand]

Operand (dst, src)
AX, #word
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The source operand (src) specified by the 2nd operand is subtracted from the destination operand (dst) specified by the 1st operand and the result is stored in the destination operand (dst) and the CY flag.
- If the subtraction shows that dst is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the subtraction generates a borrow out of bit 15, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- As a result of subtraction, the AC flag becomes undefined.

[Description example]

```
SUBW    AX, #ABCDH      ;  (1)
```

(1) ABCDH is subtracted from the AX register contents and the result is stored in the AX register.

CMPW

Word data comparison

[Instruction format]

CMPW dst, src

[Operation]

dst - src

[Operand]

Operand (dst, src)
AX, #word
AX, BC
AX, DE
AX, HL
AX, saddrp
AX, !addr16
AX, [HL+byte]
AX, ES:!addr16
AX, ES:[HL+byte]

[Flag]

Z	AC	CY
x	x	x

x : Set or cleared according to the result

[Description]

- The source operand (src) specified by the 2nd operand is subtracted from the destination operand (dst) specified by the 1st operand.
  The subtraction result is not stored anywhere and only the Z, AC and CY flags are changed.
- If the subtraction result is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the subtraction generates a borrow out of bit 15, the CY flag is set (1). In all other cases, the CY flag is cleared (0).
- As a result of subtraction, the AC flag becomes undefined.

[Description example]

```
CMPW    AX, #ABCDH              ;  (1)
```

(1) ABCDH is subtracted from the AX register and only the flags are changed.
(comparison of the AX register and the immediate data)

(5) Multiply instruction

The following multiply instructions are avaialble.

Instruction	Overview
MULU	Unsigned data multiplication

MULU

Unsigned data multiplication

[Instruction format]

MULU src

[Operation]

AX <- A x src

[Operand]

Operand (src)
X

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The A register contents and the source operand (src) data are multiplied as unsigned data and the result is stored in the AX register.

[Description example]

```
MULU    X       ;  (1)
```

(1) The A register contents and the X register contents are multiplied and the result is stored in the AX register.

÷

(6) Increment/decrement instructions

The following increment/decrement instructions are available.

Instruction	Overview
INC	Byte adta increment
DEC	Byte data decrement
INCW	Word data increment
DECW	Word data decrement

INC

Byte adta increment

[Instruction format]

INC dst

[Operation]

dst <- dst + 1

[Operand]

Operand (src)
r
saddr
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[Flag]

Z	AC	CY
x	x	

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents are incremented by only one.
- If the increment result is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the increment generates a carry for bit 4 out of bit 3, the AC flag is set (1). In all other cases, the AC flag is cleared (0).
- Because this instruction is frequently used for increment of a counter for repeated operations and an indexed addressing offset register, the CY flag contents are not changed (to hold the CY flag contents in multiple-byte operation).

[Description example]

```
INC     B               ;  (1)
```

(1) The B register is incremented.

DEC

Byte data decrement

[Instruction format]

DEC dst

[Operation]

dst <- dst - 1

[Operand]

Operand (src)
r
saddr
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[Flag]

Z	AC	CY
x	x	

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents are decremented by only one.
- If the decrement result is 0, the Z flag is set (1). In all other cases, the Z flag is cleared (0).
- If the decrement generates a carry for bit 3 out of bit 4, the AC flag is set (1). In all other cases, the AC flag is cleared (0).
- Because this instruction is frequently used for a counter for repeated operations, the CY flag contents are not changed (to hold the CY flag contents in multiple-byte operation).
- If dst is the B or C register or saddr, and it is not desired to change the AC and CY flag contents, the DBNZ instruction can be used.

[Description example]

```
DEC     FFE92H          ; (1)
```

(1) The contents at address FFE92H are decremented.

INCW

Word data increment

[Instruction format]

INCW dst

[Operation]

dst <- dst + 1

[Operand]

Operand (src)
rp
saddrp
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The destination operand (dst) contents are incremented by only one.
- Because this instruction is frequently used for increment of a register (pointer) used for addressing, the Z, AC and CY flag contents are not changed.

[Description example]

```
INCW    HL              ;  (1)
```

(1) The HL register is incremented.

DECW

Word data decrement

[Instruction format]

DECW dst

[Operation]

dst <- dst - 1

[Operand]

Operand (src)
rp
saddrp
!addr16
[HL+byte]
ES:!addr16
ES:[HL+byte]

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The destination operand (dst) contents are decremented by only one.
- Because this instruction is frequently used for decrement of a register (pointer) used for addressing, the Z, AC and CY flag contents are not changed.

[Description example]

```
DECW    DE              ;  (1)
```

(1) The DE register is decremented.

(7) Shift instructions

The following shift instructions are avaialable.

Instruction	Overview
SHR	Logical shift to the right
SHRW	Logical shift to the right
SHL	Logical shift to the left
SHLW	Logical shift to the left
SAR	Arithmetic shift to the right
SARW	Arithmetic shift to the right

SHR

Logical shift to the right

[Instruction format]

SHR dst, cnt

[Operation]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_7 <- 0) x cnt

[Operand]

Operand (dst, cnt)
A, cnt

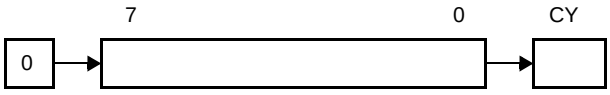
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the first operand is shifted to the right the number of times specified by cnt.
- “0” is entered to the MSB (bit 7) and the value shifted last from bit 0 is entered to CY.
- cnt can be specified as any value from 1 to 7.



[Description example]

```
SHR     A, 3    ; (1)
```

(1) When the A register's value is F5H, A = 1EH and CY = 1.

```
A = F5H  CY = 0
A = 7AH  CY = 1  1 time
A = 3DH  CY = 0  2 times
A = 1EH  CY = 1  3 times
```

SHRW

Logical shift to the right

[Instruction format]

SHRW dst, cnt

[Operation]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_{15} <- 0) x cnt

[Operand]

Operand (dst, cnt)
AX, cnt

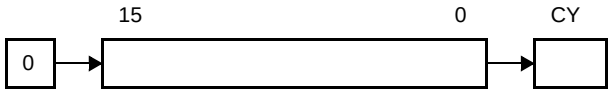
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the first operand is shifted to the right the number of times specified by cnt.
- "0" is entered to the MSB (bit 15) and the value shifted last from bit 0 is entered to CY.
- cnt can be specified as any value from 1 to 15.



[Description example]

```
SHRW    AX, 3   ; (1)
```

(1) When the AX register's value is AAF5H, AX = 155EH and CY = 1.

```
AX = AAF5H  CY = 0
AX = 557AH  CY = 1  1 time
AX = 2ABDH  CY = 0  2 times
AX = 155EH  CY = 1  3 times
```

SHL

Logical shift to the left

[Instruction format]

SHL dst, cnt

[Operation]

(CY <- dst_7, dst_m <- dst_{m-1}, dst_0 <- 0) x cnt

[Operand]

Operand (dst, cnt)
A, cnt
B, cnt
C, cnt

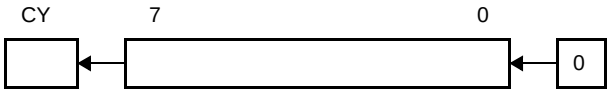
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the first operand is shifted to the left the number of times specified by cnt.
- "0" is entered to the LSB (bit 0) and the value shifted last from bit 7 is entered to CY.
- cnt can be specified as any value from 1 to 7.



[Description example]

```
SHL     A, 3    ; (1)
```

(1) When the A register's value is 5DH, A = E8H and CY = 0.

```
A = 5DH  CY = 0
A = BAH  CY = 0  1 time
A = 74H  CY = 1  2 times
A = E8H  CY = 0  3 times
```

SHLW

Logical shift to the left

[Instruction format]

SHLW dst, cnt

[Operation]

(CY <- dst_{15}, dst_m <- dst_{m-1}, dst_0 <- 0) x cnt

[Operand]

Operand (dst, cnt)
AX, cnt
BC, cnt

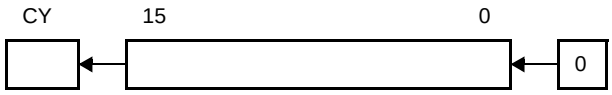
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the first operand is shifted to the left the number of times specified by cnt.
- “0” is entered to the LSB (bit 0) and the value shifted last from bit 15 is entered to CY.
- cnt can be specified as any value from 1 to 15.



[Description example]

```
SHLW    BC, 3   ; (1)
```

(1) When the BC register's value is C35DH, BC = 1AE8H and CY = 0.

```
BC = C35DH  CY = 0
BC = 86BAH  CY = 1  1 time
BC = 0D74H  CY = 1  2 times
BC = 1AE8H  CY = 0  3 times
```

SAR

Arithmetic shift to the right

[Instruction format]

SAR dst, cnt

[Operation]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_7 <- dst_7) x cnt

[Operand]

Operand (dst, cnt)
A, cnt

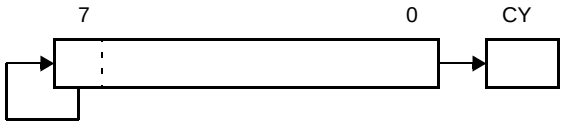
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the first operand is shifted to the right the number of times specified by cnt.
- The same value is retained in the MSB (bit 7), and the value shifted last from bit 0 is entered to CY.
- cnt can be specified as any value from 1 to 7.



[Description example]

```
SAR     A, 4    ; (1)
```

(1) When the A register's value is 8CH, A = F8H and CY = 1.

```
A = 8CH  CY = 0
A = C6H  CY = 0  1 time
A = E3H  CY = 0  2 times
A = F1H  CY = 1  3 times
A = F8H  CY = 1  4 times
```

SARW

Arithmetic shift to the right

[Instruction format]

SARW dst, cnt

[Operation]

(CY <- dst_0, dst_{m-1} <- dst_m, dst_{15} <- dst_{15}) x cnt

[Operand]

Operand (dst, cnt)
AX, cnt

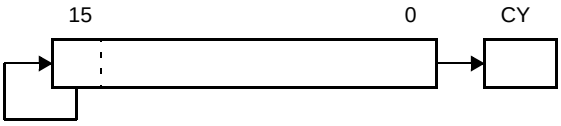
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) specified by the first operand is shifted to the right the number of times specified by cnt.
- The same value is retained in the MSB (bit 15), and the value shifted last from bit 0 is entered to CY.
- cnt can be specified as any value from 1 to 15.



[Description example]

```
SAR     AX, 4   ; (1)
```

(1) When the AX register's value is A28CH, AX = FA28H and CY = 1.

```
AX = A28CH  CY = 0
AX = D146H  CY = 0  1 time
AX = E8A3H  CY = 0  2 times
AX = F451H  CY = 1  3 times
AX = FA28H  CY = 1  4 times
```

(8) Rotate instructions

The following rotation instructions are available.

Instruction	Overview
ROR	Byte data rotation to the right
ROL	Byte data rotation to the left
RORC	Byte data rotation to the right with carry
ROLC	Byte data rotation to the left with carry
ROLWC	Word data rotation to the left with carry

ROR

Byte data rotation to the right

[Instruction format]

ROR dst, cnt

[Operation]

(CY, dst_7 <- dst_0, dst_{m-1} <- dst_m) x one time

[Operand]

Operand (dst, cnt)
A, 1

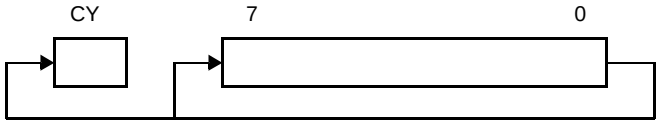
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents specified by the 1st operand are rotated to the right just once.
- The LSB (bit 0) contents are simultaneously rotated to the MSB (bit 7) and transferred to the CY flag.



[Description example]

```
ROR     A, 1          ;  (1)
```

(1) The A register contents are rotated to the right by one bit.

ROL

Byte data rotation to the left

[Instruction format]

ROL dst, cnt

[Operation]

(CY, dst_0 <- dst_7, dst_{m+1} <- dst_m) x one time

[Operand]

Operand (dst, cnt)
A, 1

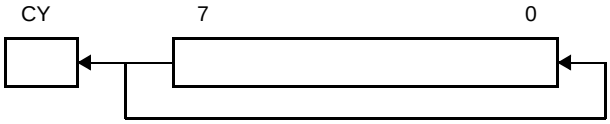
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents specified by the 1st operand are rotated to the left just once.
- The MSB (bit 7) contents are simultaneously rotated to the LSB (bit 0) and transferred to the CY flag.



[Description example]

```
ROL     A, 1          ;  (1)
```

(1) The A register contents are rotated to the left by one bit.

RORC

Byte data rotation to the right with carry

[Instruction format]

RORC dst, cnt

[Operation]

(CY <- dst_0, dst_7 <- CY, dst_{m-1} <- dst_m) x one time

[Operand]

Operand (dst, cnt)
A, 1

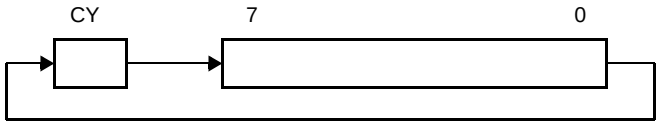
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents specified by the 1st operand are rotated just once to the right with carry.



[Description example]

```
RORC    A, 1            ;  (1)
```

(1) The A register contents are rotated to the right by one bit including the CY flag.

ROLC

Byte data rotation to the left with carry

[Instruction format]

ROLC dst, cnt

[Operation]

(CY <- dst_7, dst_0 <- CY, dst_{m+1} <- dst_m) x one time

[Operand]

Operand (dst, cnt)
A, 1

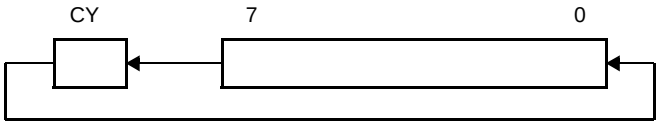
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents specified by the 1st operand are rotated just once to the left with carry.



[Description example]

```
ROLC    A, 1            ;  (1)
```

(1) The A register contents are rotated to the left by one bit including the CY flag.

ROLWC

Word data rotation to the left with carry

[Instruction format]

ROLWC dst, cnt

[Operation]

(CY <- dst_{15}, dst_0 <- CY, dst_{m+1} <- dst_m) x one time

[Operand]

Operand (dst, cnt)
AX, 1
BC, 1

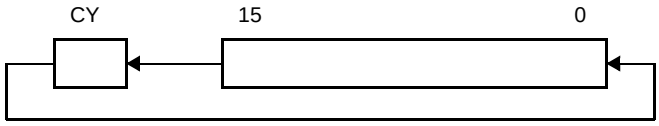
[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The destination operand (dst) contents specified by the 1st operand are rotated just once to the left with carry.



[Description example]

```
ROLWC   BC, 1          ; (1)
```

(1) The BC register contents are rotated to the left by one bit including the CY flag.

(9) Bit manipulation instructions

The following bit manipulation instructions are available.

Instruction	Overview
MOV1	1-bit data transfer
AND1	1-bit data AND operation
OR1	1-bit data OR operation
XOR1	1-bit data exclusive OR operation
SET1	1-bit data set
CLR1	1-bit data clear
NOT1	1-bit data logical negation

MOV1

1-bit data transfer

[Instruction format]

MOV1 dst, src

[Operation]

dst <- src

[Operand]

Operand (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
saddr.bit, CY
sfr.bit, CY
A.bit, CY
PSW.bit, CY
[HL].bit, CY
CY, ES:[HL].bit
ES:[HL].bit, CY

[Flag]

(1) dst = CY

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

(2) dst = PSW.bit

Z	AC	CY
x	x	

Blank : Unchanged
x : Set or cleared according to the result

(3) All other operand combinations

Z	AC	CY

Blank : Unchanged

[Description]

- Bit data of the source operand (src) specified by the 2nd operand is transferred to the destination operand (dst) specified by the 1st operand.
- When the destination operand (dst) is CY or PSW.bit, only the corresponding flag is changed.

[Description example]

```
MOV1    P3.4, CY    ;  (1)
```

(1) The CY flag contents are transferred to bit 4 of port 3.

AND1

1-bit data AND operation

[Instruction format]

AND1 dst, src

[Operation]

dst <- dst ^ src

[Operand]

Operand (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- Logical product of bit data of the destination operand (dst) specified by the 1st operand and the source operand (src) specified by the 2nd operand is obtained and the result is stored in the destination operand (dst).
- The operation result is stored in the CY flag (because of the destination operand (dst)).

[Description example]

```
AND1  CY, FFE7FH.3  ; (1)
```

(1) Logical product of FFE7FH bit 3 and the CY flag is obtained and the result is stored in the CY flag.

OR1

1-bit data OR operation

[Instruction format]

OR1 dst, src

[Operation]

dst <- dst v src

[Operand]

Operand (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The logical sum of bit data of the destination operand (dst) specified by the 1st operand and the source operand (src) specified by the 2nd operand is obtained and the result is stored in the destination operand (dst).
- The operation result is stored in the CY flag (because of the destination operand (dst)).

[Description example]

```
OR1     CY, P2.5    ;  (1)
```

(1) The logical sum of port 2 bit 5 and the CY flag is obtained and the result is stored in the CY flag.

XOR1

1-bit data exclusive OR operation

[Instruction format]

XOR1 dst, src

[Operation]

dst <- dst ∀ src

[Operand]

Operand (dst, src)
CY, saddr.bit
CY, sfr.bit
CY, A.bit
CY, PSW.bit
CY, [HL].bit
CY, ES:[HL].bit

[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The exclusive logical sum of bit data of the destination operand (dst) specified by the 1st operand and the source operand (src) specified by the 2nd operand is obtained and the result is stored in the destination operand (dst).
- The operation result is stored in the CY flag (because of the destination operand (dst)).

[Description example]

```
XOR1  CY, A.7 ;  (1)
```

(1) The exclusive logical sum of the A register bit 7 and the CY flag is obtained and the result is stored in the CY flag.

SET1

1-bit data set

[Instruction format]

SET1 dst

[Operation]

dst <- 1

[Operand]

Operand (dst)
saddr.bit
sfr.bit
A.bit
!addr16.bit
PSW.bit
[HL].bit
ES:!addr16.bit
ES:[HL].bit
CY

[Flag]

(1) dst = PSW.bit

Z	AC	CY
x	x	x

x : Set or cleared according to the result

(2) dst = CY

Z	AC	CY
		1

Blank : Unchanged
1 : Set to 1

(3) All other operand combinations

Z	AC	CY

Blank : Unchanged

[Description]

- The destination operand (dst) is set (1).
- When the destination operand (dst) is CY or PSW.bit, only the corresponding flag is set (1).

[Description example]

```
SET1    FFE55H.1        ; (1)
```

(1) Bit 1 of FFE55H is set (1).

CLR1

1-bit data clear

[Instruction format]

CLR1 dst

[Operation]

dst <- 0

[Operand]

Operand (dst)
saddr.bit
sfr.bit
A.bit
!addr16.bit
PSW.bit
[HL].bit
ES:!addr16.bit
ES:[HL].bit
CY

[Flag]

(1) dst = PSW.bit

Z	AC	CY
x	x	x

x : Set or cleared according to the result

(2) dst = CY

Z	AC	CY
		0

Blank : Unchanged
0 : Cleared to 0

(3) All other operand combinations

Z	AC	CY

Blank : Unchanged

[Description]

- The destination operand (dst) is cleared (0).
- When the destination operand (dst) is CY or PSW.bit, only the corresponding flag is cleared (0).

[Description example]

```
CLR1    P3.7            ;  (1)
```

(1) Bit 7 of port 3 is cleared (0).

NOT1

1-bit data logical negation

[Instruction format]

NOT1 dst

[Operation]

dst <- $\overline{\text{dst}}$

[Operand]

Operand (dst)
CY

[Flag]

Z	AC	CY
		x

Blank : Unchanged
x : Set or cleared according to the result

[Description]

- The CY flag is inverted.

[Description example]

```
NOT1    CY      ;  (1)
```

(1) The CY flag is inverted.

(10) Call return instructions

The following call return instructions are available.

Instruction	Overview
CALL	Subroutine call
CALLT	Subroutine call (call table reference)
BRK	Software vector interrupt
RET	Return from subroutine
RETI	Return from hardware vector interrupt
RETB	Return from software interrupt

CALL

Subroutine call

[Instruction format]

CALL target

[Operation]

(SP - 2) <- $(PC + n)_S$,
(SP - 3) <- $(PC + n)_H$,
(SP - 4) <- $(PC + n)_L$,
SP <- SP - 4,
PC <- target

Remark n is 4 when using !!addr20, 3 when using !addr16 or $!addr20, and 2 when using AX, BC, DE, or HL.

[Operand]

Operand (target)
AX
BC
DE
HL
$!addr20
!addr16
!!addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This is a subroutine call with a 20/16-bit absolute address or a register indirect address.
- The start address (PC+n) of the next instruction is saved in the stack and is branched to the address specified by the target operand (target).

[Description example]

```
CALL    !!3E000H        ; (1)
```

(1) Subroutine call to 3E000H.

CALLT

Subroutine call (call table reference)

[Instruction format]

CALLT [addr5]

[Operation]

(SP - 2) <- $(PC + 2)_S$,
(SP - 3) <- $(PC + 2)_H$,
(SP - 4) <- $(PC + 2)_L$,
PC_S <- 0000,
PC_H <- (000000000000, addr5 + 1),
PC_L <- (000000000000, addr5),
SP <- SP - 4

[Operand]

Operand ([addr5])
[addr5]

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This is a subroutine call for call table reference.
- The start address (PC+2) of the next instruction is saved in the stack and is branched to the address indicated with the word data of a call table (with the higher 12 bits of the address fixed to 000000000000B, and the lower 5 bits out of 8 bits indicated with addr5).

[Description example]

```
CALLT   [80H]          ; (1)
```

(1) Subroutine call to the word data addresses 00080H and 00081H.

[Remark]

- Only even-numbered addresses can be specified (odd-numbered addresses cannot be specified).
  addr5: Immediate data or label from 00080H to 000BEH (even-numbered addresses only)

BRK

Software vector interrupt

[Instruction format]

BRK

[Operation]

(SP - 1) <- PSW,
(SP - 2) <- (PC + 2)$_S$,
(SP - 3) <- (PC + 2)$_H$,
(SP - 4) <- (PC + 2)$_L$,
PC$_S$ <- 0000,
PC$_H$ <- (0007FH),
PC$_L$ <- (0007EH),
SP <- SP - 4,
IE<- 0

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This is a software interrupt instruction.
- PSW and the next instruction address (PC+2) are saved to the stack. After that, the IE flag is cleared (0) and the saved data is branched to the address indicated with the word data at the vector address (0007EH, 0007FH).
  Because the IE flag is cleared (0), the subsequent maskable vectored interrupts are disabled.
- The RETB instruction is used to return from the software vectored interrupt generated with this instruction.

RET

Return from subroutine

[Instruction format]

RET

[Operation]

PC_L <- (SP),
PC_H <- (SP + 1),
PC_S <- (SP + 2),
SP <- SP + 4,

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This is a return instruction from the subroutine call made with the CALL and CALLT instructions.
- The word data saved to the stack returns to the PC, and the program returns from the subroutine.

RETI

Return from hardware vector interrupt

[Instruction format]

RETI

[Operation]

PC_L <- (SP),
PC_H <- (SP + 1),
PC_S <- (SP + 2),
PSW <- (SP + 3),
SP <- SP + 4

[Operand]

None

[Flag]

Z	AC	CY
R	R	R

R : Previously saved value is restored

[Description]

- This is a return instruction from the vectored interrupt.
- The data saved to the stack returns to the PC and the PSW, and the program returns from the interrupt servicing routine.
- This instruction cannot be used for return from the software interrupt with the BRK instruction.
- None of interrupts are acknowledged between this instruction and the next instruction to be executed.
- The NMIS flag is set to 1 by the non-maskable interrupt acceptance, and cleared to 0 by the RETI instruction.

[Cautions]

- Any interrupt (including non-maskable interrupts) is not accepted because the NMIS flag is not cleared to 0 when returning from the non-maskable interrupt processing by the instructions other than the RETI instruction.

RETB

Return from software interrupt

[Instruction format]

RETB

[Operation]

PC_L <- (SP),
PC_H <- (SP + 1),
PC_S <- (SP + 2),
PSW <- (SP + 3),
SP <- SP + 4

[Operand]

None

[Flag]

Z	AC	CY
R	R	R

R : Previously saved value is restored

[Description]

- This is a return instruction from the software interrupt generated with the BRK instruction.
- The data saved in the stack returns to the PC and the PSW, and the program returns from the interrupt servicing routine.
- None of interrupts are acknowledged between this instruction and the next instruction to be executed.

(11) Stack manipulation instructions

The following stack manipulation instructions are available.

Instruction	Overview
PUSH	Push
POP	Pop
MOVW	Stck pointer and word data transfer
ADDW	Stack pointer addition
SUBW	Stack pointer subtraction

PUSH

Push

[Instruction format]

PUSH src

[Operation]

(1) src = rp

(SP - 1) <- rp_H,
(SP - 2) <- rp_L,
SP <- SP - 2

(2) src = PSW

(SP - 1) <- PSW,
(SP - 2) <- 00H,
SP <- SP - 2

[Operand]

Operand (src)
PSW
rp

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The data of the register specified by the source operand (src) is saved to the stack.

[Description example]

```
PUSH    AX              ; (1)
```

(1) AX register contents are saved to the stack.

POP

Pop

[Instruction format]

POP dst

[Operation]

(1) dst = rp

rp_L <- (SP),
rp_H <- (SP + 1),
SP <- SP + 2

(2) dst = PSW

PSW <- (SP + 1),
SP <- SP + 2

[Operand]

Operand (dst)
PSW
rp

[Flag]

(1) dst = rp

Z	AC	CY

Blank : Unchanged

(2) dst = PSW

Z	AC	CY
R	R	R

R : Previously saved value is restored

[Description]

- Data is returned from the stack to the register specified by the destination operand (dst).
- When the operand is PSW, each flag is replaced with stack data.
- None of interrupts are acknowledged between the POP PSW instruction and the subsequent instruction.

[Description example]

```
POP     AX              ;  (1)
```

(1) The stack data is returned to the AX register.

MOVW

Stck pointer and word data transfer

[Instruction format]

MOVW dst, src

[Operation]

dst <- src

[Operand]

Operand (dst, src)
SP, #word
SP, AX
AX, SP
HL, SP
BC, SP
DE, SP

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This is an instruction to manipulate the stack pointer contents.
- The source operand (src) specified by the 2nd operand is stored in the destination operand (dst) specified by the 1st operand.

[Description example]

```
MOVW    SP, #FE1FH              ;  (1)
```

(1) FE1FH is stored in the stack pointer.

ADDW

Stack pointer addition

[Instruction format]

ADDW SP, src

[Operation]

SP <- SP + src

[Operand]

Operand (SP, src)
SP, #byte

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The stack pointer specified by the first operand and the source operand (src) specified by the second operand are added and the result is stored in the stack pointer.

[Description example]

```
ADDW    SP, #12H    ; (1)
```

(1) Stack pointer and 12H are added, and the result is stored in the stack pointer.

SUBW

Stack pointer subtraction

[Instruction format]

SUBW SP, src

[Operation]

SP <- SP - src

[Operand]

Operand (SP, src)
SP, #byte

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- Source operand (src) specified by the second operand is subtracted from the stack pointer specified by the first operand, and the result is stored in the stack pointer.

[Description example]

```
SUBW    SP, #12H    ; (1)
```

(1) 12H is subtracted from the stack pointer, and the result is stored in the stack pointer.

(12) Unconditional branch instruction

The following unconditional branch instructions are available

Instruction	Overview
BR	Unconditional branch

BR

Unconditional branch

[Instruction format]

BR target

[Operation]

PC <- target

[Operand]

Operand (target)
AX
$addr20
$!addr20
!addr16
!!addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This is an instruction to branch unconditionally.
- The word data of the target address operand (target) is transferred to PC and branched.

[Description example]

```
BR      !!12345H        ; (1)
```

(1) Branch to address 12345H.

(13) Conditional branch instructions

The following conditional branch instructions are available.

Instruction	Overview
BC	Conditional branch with carry flag (CY = 1)
BNC	Conditional branch with carry flag (CY = 0)
BZ	Conditional branch with zero flag (Z = 1)
BNZ	Conditional branch with zero flag (Z = 0)
BH	Conditional branch by numerical size ((Z v CY) = 0)
BNH	Conditional branch by numerical size ((Z v CY) = 1)
BT	Conditional branch by bit test (byte date bit = 1)
BF	Conditional branch by bit test (byte date bit = 0)
BTCLR	Conditional branch and clear by bit test (byte date bit = 1)

BC

Conditional branch with carry flag (CY = 1)

[Instruction format]

BC $addr20

[Operation]

PC <- PC + 2 + jdisp8 if CY = 1

[Operand]

Operand ($addr20)
$addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When CY = 1, data is branched to the address specified by the operand.
  When CY = 0, no processing is carried out and the subsequent instruction is executed.

[Description example]

```
BC      $00300H         ; (1)
```

(1) When CY = 1, data is branched to 00300H (with the start of this instruction set in the range of addresses 0027FH to 0037EH).

BNC

Conditional branch with carry flag (CY = 0)

[Instruction format]

BNC $addr20

[Operation]

PC <- PC + 2 + jdisp8 if CY = 0

[Operand]

Operand ($addr20)
$addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When CY = 0, data is branched to the address specified by the operand.
  When CY = 1, no processing is carried out and the subsequent instruction is executed.

[Description example]

```
BNC     $00300H         ; (1)
```

(1) When CY = 0, data is branched to 00300H (with the start of this instruction set in the range of addresses 0027FH to 0037EH).

BZ

Conditional branch with zero flag (Z = 1)

[Instruction format]

BZ $addr20

[Operation]

PC <- PC + 2 + jdisp8 if Z = 1

[Operand]

Operand ($addr20)
$addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When Z = 1, data is branched to the address specified by the operand.
  When Z = 0, no processing is carried out and the subsequent instruction is executed.

[Description example]

```
DEC B
BZ      $003C5H         ; (1)
```

(1) When the B register is 0, data is branched to 003C5H (with the start of this instruction set in the range of addresses 00344H to 00443H).

BNZ

Conditional branch with zero flag (Z = 0)

[Instruction format]

BNZ $addr20

[Operation]

PC <- PC + 2 + jdisp8 if Z = 0

[Operand]

Operand ($addr20)
$addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When Z = 0, data is branched to the address specified by the operand.
  When Z = 1, no processing is carried out and the subsequent instruction is executed.

[Description example]

```
CMP     A, #55H
BNZ     $00A39H         ; (1)
```

(1) If the A register is not 55H, data is branched to 00A39H (with the start of this instruction set in the range of addresses 009B8H to 00AB7H).

BH

Conditional branch by numerical size ((Z v CY) = 0)

[Instruction format]

BH $addr20

[Operation]

PC <- PC + 3 + jdisp8 if (Z v CY) = 0

[Operand]

Operand ($addr20)
$addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When (Z v CY) = 0, data is branched to the address specified by the operand.
  When (Z v CY) = 1, no processing is carried out and the subsequent instruction is executed.
- This instruction is used to judge which of the unsigned data values is higher. It is detected whether the first operand is higher than the second operand in the CMP instruction immediately before this instruction.

[Description example]

```
CMP     A, C
BH      $00356H     ; (1)
```

(1) Branch to address 00356H when the A register contents are greater than the C register contents (start of the BH instruction, however, is in addresses 002D4H to 003D3H).

BNH

Conditional branch by numerical size ((Z v CY) = 1)

[Instruction format]

BNH $addr20

[Operation]

PC <- PC + 3 + jdisp8 if (Z v CY) = 1

[Operand]

Operand ($addr20)
$addr20

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When (Z v CY) = 1, data is branched to the address specified by the operand.
  When (Z v CY) = 0, no processing is carried out and the subsequent instruction is executed.
- This instruction is used to judge which of the unsigned data values is higher. It is detected whether the first operand is not higher than the second operand (the first operand is equal to or lower than the second operand) in the CMP instruction immediately before this instruction.

[Description example]

```
CMP     A, C
BNH     $00356H     ; (1)
```

(1) Branch to address 00356H when the A register contents are equal to or lower than the C register contents (start of the BNH instruction, however, is in addresses 002D4H to 003D3H).

BT

Conditional branch by bit test (byte date bit = 1)

[Instruction format]

BT bit, $addr20

[Operation]

PC <- PC + b + jdisp8 if bit = 1

[Operand]

Operand (bit, $addr20)	b (Number of Bytes)
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- If the 1st operand (bit) contents have been set (1), data is branched to the address specified by the 2nd operand ($addr20).
  If the 1st operand (bit) contents have not been set (1), no processing is carried out and the subsequent instruction is executed.

[Description example]

```
BT      FFE47H.3, $0055CH   ; (1)
```

(1) When bit 3 at address FFE47H is 1, data is branched to 0055CH (with the start of this instruction set in the range of addresses 004DAH to 005D9H).

BF

Conditional branch by bit test (byte date bit = 0)

[Instruction format]

BF bit, $addr20

[Operation]

PC <- PC + b + jdisp8 if bit = 0

[Operand]

Operand (bit, $addr20)	b (Number of Bytes)
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- If the 1st operand (bit) contents have been cleared (0), data is branched to the address specified by the 2nd operand ($addr20).
  If the 1st operand (bit) contents have not been cleared (0), no processing is carried out and the subsequent instruction is executed.

[Description example]

```
BF      P2.2, $01549H       ; (1)
```

(1) When bit 2 of port 2 is 0, data is branched to address 01549H (with the start of this instruction set in the range of addresses 014C6H to 015C5H).

BTCLR

Conditional branch and clear by bit test (byte date bit = 1)

[Instruction format]

BTCLR bit, $addr20

[Operation]

PC <- PC + b + jdisp8 if bit = 1, then bit <- 0

[Operand]

Operand (bit, $addr20)	b (Number of Bytes)
saddr.bit, $addr20	4
sfr.bit, $addr20	4
A.bit, $addr20	3
PSW.bit, $addr20	4
[HL].bit, $addr20	3
ES:[HL].bit, $addr20	4

[Flag]

(1) bit = PSW.bit

Z	AC	CY
x	x	x

x : Set or cleared according to the result

(2) All other operand combinations

Z	AC	CY

Blank : Unchanged

[Description]

- If the 1st operand (bit) contents have been set (1), they are cleared (0) and branched to the address specified by the 2nd operand.
  If the 1st operand (bit) contents have not been set (1), no processing is carried out and the subsequent instruction is executed.
- When the 1st operand (bit) is PSW.bit, the corresponding flag contents are cleared (0).

[Description example]

```
BTCLR PSW.0, $00356H    ; (1)
```

(1) When bit 0 (CY flag) of PSW is 1, the CY flag is cleared to 0 and branched to address 00356H (with the start of this instruction set in the range of addresses 002D4H to 003D3H).

(14) Conditional skip instructions

The following conditional skip instructions are available.

Instruction	Overview
SKC	Skip with carry flag (CY = 1)
SKNC	Skip with carry flag (CY = 0)
SKZ	Skip with zero flag (Z = 1)
SKNZ	Skip with zero flag (Z = 0)
SKH	Skip by numerical size ((Z v CY) = 0)
SKNH	Skip by numerical size ((Z v CY) =1)

SKC

Skip with carry flag (CY = 1)

[Instruction format]

SKC

[Operation]

Next instruction skip if CY = 1

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When CY = 1, the next instruction is skipped. The subsequent instruction is a NOP and one clock of execution time is consumed. However, if the next instruction is a PREFIX instruction (indicated by "ES:"), two clocks of execution time are consumed.
- When CY = 0, the next instruction is executed.

[Description example]

```
MOV     A, #55H
SKC
ADD     A, #55H     ; (1)
```

(1) The A register's value = AAH when CY = 0, and 55H when CY = 1.

SKNC

Skip with carry flag (CY = 0)

[Instruction format]

SKNC

[Operation]

Next instruction skip if CY = 0

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When CY = 0, the next instruction is skipped. The subsequent instruction is a NOP and one clock of execution time is consumed. However, if the next instruction is a PREFIX instruction (indicated by "ES:"), two clocks of execution time are consumed.
- When CY = 1, the next instruction is executed.

[Description example]

```
MOV     A, #55H
SKNC
ADD     A, #55H     ; (1)
```

(1) The A register's value = AAH when CY = 1, and 55H when CY = 0.

SKZ

Skip with zero flag (Z = 1)

[Instruction format]

SKZ

[Operation]

Next instruction skip if Z = 1

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When Z = 1, the next instruction is skipped. The subsequent instruction is a NOP and one clock of execution time is consumed. However, if the next instruction is a PREFIX instruction (indicated by "ES:"), two clocks of execution time are consumed.
- When Z = 0, the next instruction is executed.

[Description example]

```
MOV     A, #55H
SKZ
ADD     A, #55H     ; (1)
```

(1) The A register's value = AAH when Z = 0, and 55H when Z = 1.

SKNZ

Skip with zero flag (Z = 0)

[Instruction format]

SKNZ

[Operation]

Next instruction skip if Z = 0

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When Z = 0, the next instruction is skipped. The subsequent instruction is a NOP and one clock of execution time is consumed. However, if the next instruction is a PREFIX instruction (indicated by “ES:”), two clocks of execution time are consumed.
- When Z = 1, the next instruction is executed.

[Description example]

```
MOV     A, #55H
SKNZ
ADD     A, #55H     ; (1)
```

(1) The A register’s value = AAH when Z = 1, and 55H when Z = 0.

SKH

Skip by numerical size ((Z v CY) = 0)

[Instruction format]

SKH

[Operation]

Next instruction skip if (Z v CY) = 0

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When (Z v CY) = 0, the next instruction is skipped. The subsequent instruction is a NOP and one clock of execution time is consumed. However, if the next instruction is a PREFIX instruction (indicated by "ES:"), two clocks of execution time are consumed.
- When (Z v CY) = 1, the next instruction is executed.

[Description example]

```
CMP     A, #80H
SKH
CALL    !!TARGET        ; (1)
```

(1) When the A register contents are higher than 80H, the CALL instruction is skipped and the next instruction is executed. When the A register contents are 80H or lower, the next CALL instruction is executed and execution is branched to the target address.

SKNH

Skip by numerical size ((Z v CY) =1)

[Instruction format]

SKNH

[Operation]

Next instruction skip if (Z v CY) = 1

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- When (Z v CY) = 1, the next instruction is skipped. The subsequent instruction is a NOP and one clock of execution time is consumed. However, if the next instruction is a PREFIX instruction (indicated by ES:), two clocks of execution time are consumed.
- When (Z v CY) = 0, the next instruction is executed.

[Description example]

```
CMP     A, #80H
SKNH
CALL    !!TARGET        ; (1)
```

(1) When the A register contents are 80H or lower, the CALL instruction is skipped and the next instruction is executed. When the A register contents are higher than 80H, the next CALL instruction is executed and execution is branched to the target address.

(15) CPU control instructions

The following CPU control instructions are available.

Instruction	Overview
SEL	Register bank selection
NOP	No operation
EI	Interrupt enabled
DI	Interrupt disabled
HALT	Halt mode set
STOP	Stop mode set

SEL

Register bank selection

[Instruction format]

SEL RBn

[Operation]

RBS0, RBS1 <- n ; (n = 0 to 3)

[Operand]

Operand (RBn)
RBn

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The register bank specified by the operand (RBn) is made a register bank for use by the next and subsequent instructions.
- RBn ranges from RB0 to RB3.

[Description example]

```
SEL     RB2     ; (1)
```

(1) Register bank 2 is selected as the register bank for use by the next and subsequent instructions.

NOP

No operation

[Instruction format]

NOP

[Operation]

no operation

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- Only the time is consumed without processing.

EI

Interrupt enabled

[Instruction format]

EI

[Operation]

IE <- 1

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- The maskable interrupt acknowledgeable status is set (by setting the interrupt enable flag (IE) to (1)).
- No interrupts are acknowledged between this instruction and the next instruction.
- If this instruction is executed, vectored interrupt acknowledgment from another source can be disabled. For details, see the description of interrupt functions in the user's manual for each product.

DI

Interrupt disabled

[Instruction format]

DI

[Operation]

IE <- 0

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- Maskable interrupt acknowledgment by vectored interrupt is disabled (with the interrupt enable flag (IE) cleared (0)).
- No interrupts are acknowledged between this instruction and the next instruction.
- For details of interrupt servicing, see the description of interrupt functions in the user's manual for each product.

HALT

Halt mode set

[Instruction format]

HALT

[Operation]

Set HALT Mode

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

- This instruction is used to set the HALT mode to stop the CPU operation clock. The total power consumption of the system can be decreased with intermittent operation by combining this mode with the normal operation mode.

STOP

Stop mode set

[Instruction format]

STOP

[Operation]

Set STOP Mode

[Operand]

None

[Flag]

Z	AC	CY

Blank : Unchanged

[Description]

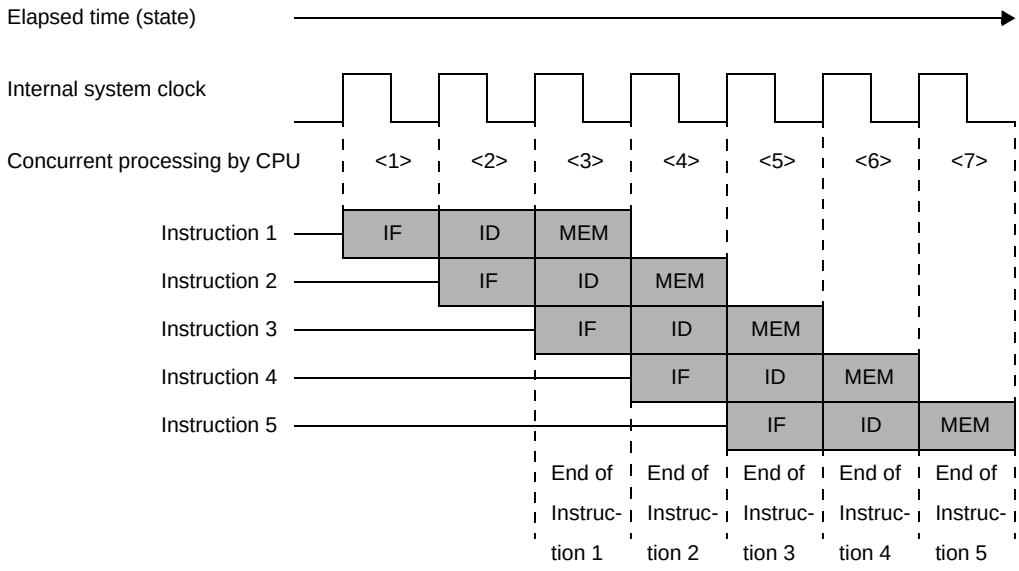
- This instruction is used to set the STOP mode to stop the main system clock oscillator and to stop the whole system. Power consumption can be minimized to only leakage current.

4.6.7 Pipeline

(1) Features

78K0R microcontrollers use three-stage pipeline control to enable single-cycle execution of almost all instructions. Instructions are executed in three stages: instruction fetch (IF), instruction decode (ID), and memory access (MEM).

Figure 4-38. Pipeline Execution of Five Typical Instructions (Example)



IF (instruction fetch)	Instruction is fetched and fetch pointer is incremented.
ID (instruction decode)	Instruction is decoded and address is calculated.
MEM (memory access)	Decoded instruction is executed and memory at target address is accessed.

(2) Number of operation clocks

An inherent problem in some microcontroller pipeline architectures is that it is impossible to predict the number of clocks required for instruction execution. In the 78K0R microcontroller, this problem has been solved. Instructions always execute in the same number of clocks, allowing stable program execution.
Except in the cases listed below, the number of execution clocks is as shown in "(5) Operation list".

(a) Access to flash memory contents as data

When the contents of flash memory are accessed as data, the pipeline stalls at the MEM stage. This adds a certain number of clocks to the number shown in the operation list. For details, see "(5) Operation list".

(b) Access to external memory contents as data

A CPU wait occurs when the content of external memory is accessed as data. This adds a certain number of clocks to the number shown in the operation list.
The following table lists the number of increased clocks.

External Expansion Clock Output (CLKOUT) Selection Clocks	Wait Cycles
f_{CLK}	3 clocks
$f_{CLK}/2$	5 or 6 clocks
$f_{CLK}/3$	7 to 9 clocks
$f_{CLK}/4$	9 to 12 clocks

(c) Instruction fetch from RAM

When instructions are fetched from RAM, the instruction queue empties because reading from RAM is late. The CPU must wait until instructions are ready in the queue. The CPU also waits if RAM is accessed during instruction fetching from RAM.

(d) Instruction fetch from external memory

When instructions are fetched from external memory, the instruction queue empties because reading from external memory is late. The CPU must wait until instructions are ready in the queue. The CPU also waits if external memory is accessed during instruction fetching from external memory.
The following table lists the number of increased clocks

External Expansion Clock Output (CLKOUT) Selection Clocks	Wait Cycles
f_{CLK}	3 clocks
$f_{CLK}/2$	5 or 6 clocks
$f_{CLK}/3$	7 to 9 clocks
$f_{CLK}/4$	9 to 12 clocks

(e) Hazards related to instruction combinations

A 1-clock wait occurs when a register is used for indirect memory access immediately after data has been written to that register.

Register	Previous Instruction	Operand of Next Instruction, or Next Instruction
DE	Write instruction to D register[Note] Write instruction to E register[Note] Write instruction to DE register[Note] SEL RBn	[DE], [DE+byte]
HL	Write instruction to H register[Note] Write instruction to L register[Note] Write instruction to HL register[Note] SEL RBn	[HL], [HL+byte], [HL+B], [HL+C], [HL].bit
B	Write instruction to B register[Note] SEL RBn	word[B], [HL+B]
C	Write instruction to C register[Note] SEL RBn	word[C], [HL+C]
BC	Write instruction to B register[Note] Write instruction to C register[Note] Write instruction to BC register[Note] SEL RBn	word[BC], [HL+B], [HL+C]

SP	MOVW SP, #word
	MOVW SP, AX
	ADDW SP, #byte
	SUBW SP, #byte
		[SP+byte]
		CALL instruction, CALLT instruction, BRK instruction, SOFT instruction, RET instruction, RETI instruction, RETB instruction, interrupt, PUSH instruction, POP instruction

CS	MOV CS, #byte MOV CS, A	CALL rp BR AX
AX	Write instruction to A register[Note] Write instruction to X register[Note] Write instruction to AX register[Note] SEL RBn	BR AX

Register	Previous Instruction	Operand of Next Instruction, or Next Instruction
AX	Write instruction to A register[Note]	CALL rp
BC	Write instruction to X register[Note]	
DE	Write instruction to B register[Note]	
HL	Write instruction to C register[Note]	
	Write instruction to D register[Note]	
	Write instruction to E register[Note]	
	Write instruction to H register[Note]	
	Write instruction to L register[Note]	
	Write instruction to AX register[Note]	
	Write instruction to BC register[Note]	
	Write instruction to DE register[Note]	
	Write instruction to HL register[Note]	
	SEL RBn	

Note Write instructions to register also require wait insertions when overwriting the target register values by direct addressing, short direct addressing, register indirect addressing, based addressing, or based indexed addressing.

CHAPTER 5 LINK DIRECTIVE SPECIFICATION

This chapter explains the necessary items for link directives and how to write a directive file.

5.1 Coding Method

This section explains coding method of link directives.

5.1.1 Link directives

Link directives (referred to as directives from here on) are a group of commands for performing various directions during linking such as input files for the linker, useable memory areas, and segment location.

There are the following two kinds of directives.

Directive Type	Purpose
Memory directive	- Declares a installed memory address. - Divides memory into a number of areas and specifies the memory area. CALLT area Internal ROM External ROM SADDR Internal RAM other than SADDR
Segment location directive	- Specifies a segment location. Specifies the following contents for each segment. Absolute address Specify only the memory area

Create a file (directive file) containing directives using a text editor and specify the -d option when starting the linker. The linker will read the directive file and perform link processing while it interprets the file.

(1) Directive file

The format for specifying directives in the directive file is shown next.

- Memory directive

```
MEMORY  memory-area-name: (start-address-value, size) [ /memory-space-name]
```

- Segment Allocation Directives

```
MERGE segment-name: [ AT∆(∆start-address∆) ] [ =memory-area-name-specification] [ /
memory-space-name]

MERGE  segment-name: [ merge-attribute ] [ =memory-area-name-specification] [ /
memory-space-name]
```

In addition, multiple directives can be specified in a single directive file.

For details about each directive, see "(2) Memory directive" and "(3) Segment location directive".

(a) Symbols

There is a distinction between uppercase and lowercase in the segment name, memory area name, and memory space name.

(b) Numerical values

When specifying numerical constants for the items in each directive, they can be specified as decimal or hexadecimal numbers.

Specifying numbers is the same as in source code, for hexadecimal numbers attach an "H" to the end of the number. Also, when the first digit is A - F, add a "0" to the beginning of the number.

Examples are shown below.

```
23H , 0FC80H
```

(c) Comments

When ";" or "#" is specified in a directive file, the text from that character to the line feed (LF) is treated as a comment. In addition, if the directive file ends before a line feed appears, the text up to the end of the file is treated as a comment.

Examples are shown below.

The underlined portions are comments.

```
; DIRECTIVE FILE FOR 78F1166 A0
MEMORY MEM1 : ( 40000H, 10000H ) #SECOND MEMORY AREA
```

(2) Memory directive

The memory directive is a directive to define a memory area (the address of memory to implement and its name).

The defined memory area can be referenced by the segment location directive using this name (memory area name).

Up to 100 memory areas can be defined, including the memory area defined by default.

The syntax is shown below.

```
MEMORY△memory-area-name△:△(△start-address△,△size△)[/△memory-space-name]
```

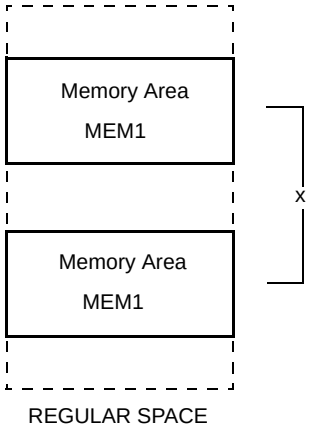
(a) Memory area name

Specifies the name of the memory area to define.

Conditions when specifying are as follows.

- Characters that can be used in the memory area name are A - Z, a - z, 0 - 9, _, ?, @.
  However, 0 - 9 cannot be used as the first character of the memory area name.
- Uppercase and lowercase characters are distinguished as separate characters.
- Uppercase and lowercase characters can be mixed together.
- The length of the memory area name is a maximum of 256 characters.
  257characters or more will result in an error.
- There must be a only a single memory area name for each one throughout all memory spaces.
  Attaching the same memory area name to differing memory areas, when the memory space is the same or when it is different, is not permitted.

Figure 5-1. Example of Memory Area Name that Cannot Be Specified



(b) Start address

Specifies the start address of the memory area to define.
Write as a numerical constant between 0H - 0FFFFFH.

(c) Size

Specifies the size of the memory area to define.

Conditions when specifying are as follows.

- A numerical constant 1 or greater.
- When re-specifying the size of the memory area the linker defines by default, there is a restriction in the range that can be defined.
  For the size of the memory area defined by default for each device and the range it can be redefined, see each device file's "Notes on Use".

(d) Memory area name

The memory space name is represented as the 64 kB space REGULAR.
Conditions when specifying are shown next.

- The memory space name is specified in all uppercase.
- When the memory space name is omitted, the linker will consider REGULAR to have been specified.
- If the memory space name is omitted after "/" is written, an error occurs.

The function is shown below.

- The memory area with the name specified by the memory area name is defined in the specified memory space.
- 1 memory area can be defined with 1 memory directive.
- There can be multiple memory directives themselves. When there multiple definitions in the specified order an error will result.
- The default memory area is valid so long as the same memory area is not redefined by a memory directive. When memory directives are omitted, the linker specifies only the default memory area for each device.
- When not using the default memory space and using it with a different area name, set the default area name's size to "0".

A usage example is shown below.

- Defines memory space address 0H to 1FFH as memory area ROMA.

```
MEMORY ROMA : ( 0H, 200H )
```

(3) Segment location directive

The segment location directive is a directive which places a specified segment in a specified memory area or places it at specific address.

The syntax is shown below.

```
MERGE∆segment-name∆:∆[AT∆(∆start-address∆)][∆=∆memory-area-name][∆/∆memory-space-name]
MERGE∆segment-name∆:∆[merge-attribute][∆=∆memory-area-name][∆/∆memory-space-name]
```

(a) Segment name

The segment name contained in the object module file input to the linker.

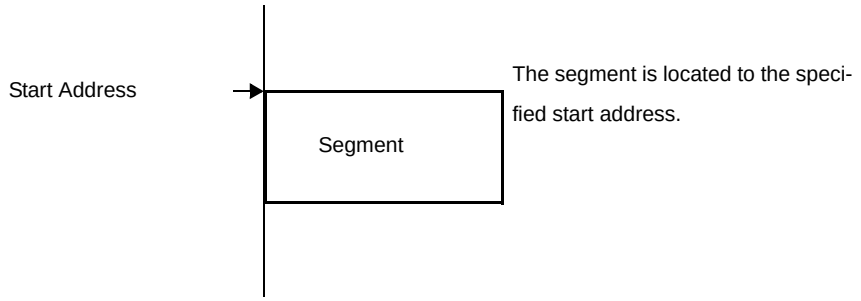
- A segment name other than the input segment cannot be specified.
- The segment name must be specified as specified in the assembler source.

(b) Start address

Allocates the segment at the area specified by the "start address".

- The reserved word AT must be specified in all uppercase or lowercase characters. It cannot mix uppercase and lowercase characters.
- The start address is specified as a numerical constant.

Figure 5-2. Specified Start Address and Segment Location



Cautions 1. When locating a segment according to the specified start address, if the range of the memory area where it is being located is exceeded, an error will result.

2. The start address cannot be specified by a link directive for segments specified a location address by the segment directive AT assignment or the ORG directive.

(c) Merge attribute

When there are multiple segments with the same name in the source, specify in the directive "COMPLETE" to error without merging or "SEQUENT (default)" to merge.

SEQUENT	Merge sequentially in the order the segments appear so as to not leave free space. BSEG is merged in bit units in the order the segments appear.
COMPLETE	An error will occur if there are multiple segments with the same name.

Examples are shown below.

```
MERGE DSEG1 : COMPLETE = RAM
```

(d) Memory area name

The memory space name specifies a memory space to locate a segment.

- Only REGULAR can be specified as the memory space name.
- The memory space name is specified in all uppercase.
- When the memory space name is omitted, the linker will consider REGULAR to have been specified.

Segment location is shown next.

Memory Area	Memory Space	Segment Location
Not specified	Not specified	The memory area located in REGULAR space at the default
Memory Area Name	Not specified	The specified memory area in REGULAR space

This table emphasizes declaring the memory area which will be the target for the segment location. In addition, if "AT(start address)" is specified when the actual location address is decided, the segment will be located from that address.
For example, for a segment with a relocation attribute of "CSEG BASE", if memory name "REGULAR" is specified, the segment will be located so that it takes up position within 0000H to FFFFH.

Notes are shown below.

- Input segments that are not specified with the segment location directive have their location address determined according to the relocation attribute specified with the segment definition directive during assembly.
- If a segment doesn't exist that is specified as a segment name, an error will result.
- If multiple segment location directives are specified for the same segment, an error will result.

5.2 Reserved Words

Reserved words for use in the directive file are shown next.
Reserved words in the directive file cannot be used for other purposes (segment names, memory area names, etc.).

Reserved Words	Explanation
MEMORY	Specifies the memory directives
MERGE	Specifies the segment location directive
AT	Specifies the location attribute (start address) of the segment location directive
SEQUENT	Specifies the merge attribute (merges a segment) of the segment location directive
COMPLETE	Specifies the merge attribute (does not merge a segment) of the segment location directive

Caution Reserved words can be specified in uppercase or lowercase. However, it cannot mix uppercase and lowercase characters.

Example MEMORY : Acceptable
memory : Acceptable
Memory : Not acceptable

5.3 Coding Examples

Link directive coding examples are shown next.

5.3.1 When specifying link directive

- An address is allocated for segment SEG1 with a segment type, relocation attribute of "CSEG UNIT". The declared memory area is as follows.

```
MEMORY ROM  : ( 0000H, 1000H )
MEMORY MEM1 : ( 1000H, 2000H )
```

- When allocating input segment SEG1 to 500H inside area ROM (see the following diagram(1)).

```
MERGE  SEG1 : AT ( 500H )
```

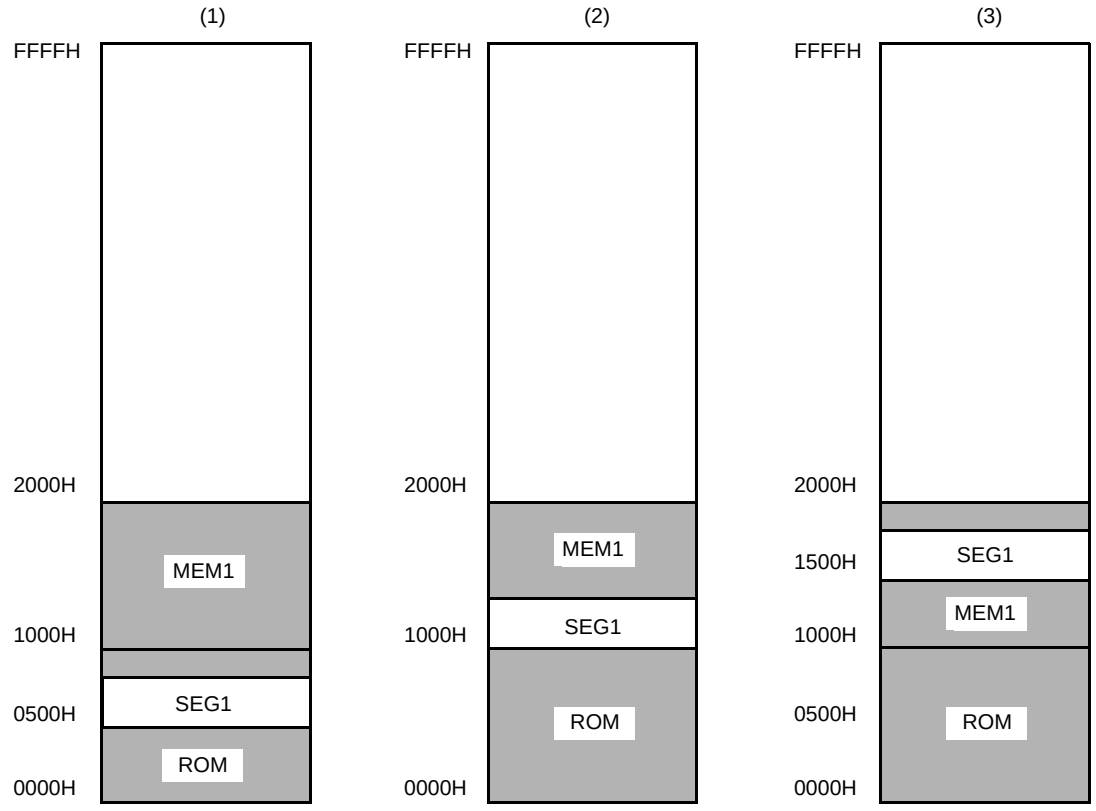
- When allocating input segment SEG1 inside memory area MEM1 (see the following diagram(2)).

```
MERGE  SEG1 : = MEM1
```

- When allocating input segment SEG1 to 1500H inside memory area MEM1 (see the following diagram(3)).

```
MERGE SEG1 : AT (1500H)=MEM178
```

Figure 5-3. Example Allocations of Input Segment SEG1



5.3.2 When using compiler

This section explains how to create a link directive file when using the compiler. Create the file matched to the actual target system and specify the created file with the -d option when linking.

Additionally, be aware of the following cautions when creating the file.

- 78K0R C compiler may use a portion of the short direct address area (saddr area) for specific purposes as shown next.
  Specifically, the 44 byte area FFEB4H to FFEDFH.

(a) When the -qr option is specified, register variables [FFEB4H to FFEC3H]

(b) norec function arguments, automatic variables [FFEC4H to FFED3H]

(c) Segment information [FFED4H to FFED7H]

(d) Runtime library arguments [FFED8H to FFEDFH]

(e) For standard library operation (a portion of areas (a) and (b))

Caution When the user is not using the standard library, area (e) is not used.

The following shows an example changing the RAM size with a link directive file (lk78k0r.dr).

When changing the memory size, be careful so that the memory does not overlap with other areas. When making changes, see the memory map of the target device to use.

```
                Start address size
memory RAM :    ( 0FCF00h, 002F20h )    -> Make this size bigger.
memory SDR :    ( 0FFE20h, 000098h )       (Also change the start address as necessary.)
merge @@INIS : = SDR                    -> Specifying the segment location.
merge @@DATS : = SDR                    -> Specifying the segment location.
merge @@BITS : = SDR                    -> Specifying the segment location.
```

When you want to change location of a segment, add the merge statement.. When the function to change the compiler output section name is used, segments can be located individually (for details see "Changing compiler output section name (#pragma section ...)").

The result of changing the segment location, when there is insufficient memory to locate the segment, change the corresponding memory statement.

CHAPTER 6 FUNCTION SPECIFICATION

In the C language there are no commands to perform input/output with external (peripheral) devices or equipment. This is because the designers of the C language designed it to keep its functions to a minimum. However, input/output operations are necessary in the development of actual systems. For this reason, library functions for performing input/output operations have been prepared in 78K0R C compiler.

This chapter explains the library functions that 78K0R C compiler has and the functions that can be used with the simulator.

6.1 Distribution Libraries

The libraries distributed with 78K0R C compiler are described below.

When using the standard library inside an application, include the relevant header files and use the library functions.

The runtime library is a portion of the standard library, but the routines are called by 78K0R C compiler automatically, they are not functions described in C or assembly language source code.

Table 6-1. Distribution Libraries

Library Type	Included Functions
Standard library	- Character/String Functions - Program Control Functions - Special Functions - Input and Output Functions - Utility Functions - String and Memory Functions - Mathematical Functions - Diagnostic Function
Runtime library	- Increment - Decrement - Sign reverse - 1's complement - Logical negation - Multiplication - Division - Remainder arithmetic - Addition - Subtraction - Left shift - Right shift - Compare - Bit AND - Bit OR - Bit XOR - Conversion from floating point - Conversion to floating point - Conversion from bit - Startup routine - Function pre- and post-processing - BCD-type conversion - Auxiliary

6.1.1 Standard library

This section shows the functions included in the standard library.

The standard library is fully supported when the -zf option is specified.

(1) Character/String Functions

Function Name	Purpose	Header File	Re-entrant
isalpha	Judges if a character is an alphabetic character (A to Z, a to z)	ctype.h	OK
isupper	Judges if a character is an uppercase alphabetic character (A to Z)	ctype.h	OK
islower	Judges if a character is an lowercase alphabetic character (a to z)	ctype.h	OK
isdigit	Judges if a character is a numeric (0 to 9)	ctype.h	OK
isalnum	Judges if a character is an alphanumeric character (0 to 9, A to Z, a to z)	ctype.h	OK
isxdigit	Judges if a character is a heNGadecimal numbers (0 to 9, A to F, a to f)	ctype.h	OK
isspace	Judges if a character is a whitespace character (whitespace, tab, carriage return, line feed, vertical, tab, form feed)	ctype.h	OK
ispunct	Judges if a character is a printable character other than a whitespace character or alphanumeric character	ctype.h	OK
isprint	Judges if a character is a printable character	ctype.h	OK
isgraph	Judges if a character is a printable character other than whitespace	ctype.h	OK
iscntrl	Judges if a character is a control character	ctype.h	OK
isascii	Judges if a character is an ASCII code	ctype.h	OK
toupper	Converts a lowercase alphabetic character to uppercase	ctype.h	OK
tolower	Converts an uppercase alphabetic character to lowercase	ctype.h	OK
toascii	Converts the input to an ASCII code	ctype.h	OK
_toupper	Subtracts "a" from the input character and adds "A"	ctype.h	OK
toup		ctype.h	OK
_tolower	Subtracts "A" from the input character and adds "a"	ctype.h	OK
tolow		ctype.h	OK

OK : Re-entrant

(2) Program Control Functions

Function Name	Purpose	Header File	Re-entrant
setjmp	Saves the environment at the time of the call	setjmp.h	NG
longjmp	Restores the environment saved with setjmp	setjmp.h	NG

NG : Not re-entrant

(3) Special Functions

Function Name	Purpose	Header File	Re-entrant
va_start	Settings for processing variable arguments	stdarg.h	OK
va_starttop	Settings for processing variable arguments	stdarg.h	OK
va_arg	Processes variable arguments	stdarg.h	OK
va_end	Indicates the end of processing variable arguments	stdarg.h	OK

OK : Re-entrant

(4) Input and Output Functions

Function Name	Purpose	Header File	Re-entrant
sprintf	Writes data to a string according to a format	stdio.h	Δ
sscanf	Reads data from the input string according to a format	stdio.h	Δ
printf	Outputs data to SFR according to a format	stdio.h	Δ
scanf	Reads data from SFR according to a format	stdio.h	Δ
vprintf	Outputs data to SFR according to a format	stdio.h	Δ
vsprintf	Writes data to a string according to a format	stdio.h	Δ
getchar	Reads one character from SFR	stdio.h	OK
gets	Reads a string	stdio.h	OK
putchar	Outputs one character to SFR	stdio.h	OK
puts	Outputs a string	stdio.h	OK
__putc	Outputs one character to opaque	stdio.h	OK

OK : Re-entrant
Δ : Functions that do not support floating point are re-entrant

(5) Utility Functions

Function Name	Purpose	Header File	Re-entrant
atoi	Converts a decimal integer string to int	stdlib.h	OK
atol	Converts a decimal integer string to long	stdlib.h	OK
strtol	Converts a string to long	stdlib.h	OK
strtoul	Converts a string to unsigned long	stdlib.h	OK
calloc	Allocates an array's region and initializes it to zero	stdlib.h	OK
free	Releases a block of allocated memory	stdlib.h	OK
malloc	Allocates a block	stdlib.h	OK
realloc	Re-allocates a block	stdlib.h	OK
abort	Abnormally terminates the program	stdlib.h	OK
atexit	Registers a function to be called at normal termination	stdlib.h	NG
exit	Terminates the program	stdlib.h	NG
abs	Obtains the absolute value of an int type value	stdlib.h	OK
labs	Obtains the absolute value of a long type value	stdlib.h	OK
div	Performs int type division, obtains the quotient and remainder	stdlib.h	NG
ldiv	Performs long type division, obtains the quotient and remainder	stdlib.h	NG
brk	Sets the break value	stdlib.h	NG
sbrk	Increases/decreases the break value	stdlib.h	NG
atof	Converts a decimal integer string to double	stdlib.h	NG
strtod	Converts a string to double	stdlib.h	NG
itoa	Converts int to a string	stdlib.h	OK
ltoa	Converts long to a string	stdlib.h	OK
ultoa	Converts unsigned long to a string	stdlib.h	OK
rand	Generates a pseudo-random number	stdlib.h	NG
srand	Initializes the pseudo-random number generator state	stdlib.h	NG
bsearch	Binary search	stdlib.h	OK
qsort	Quick sort	stdlib.h	OK
strbrk	Sets the break value	stdlib.h	OK
strsbrk	Increases/decreases the break value	stdlib.h	OK
stritoa	Converts int to a string	stdlib.h	OK
strltoa	Converts long to a string	stdlib.h	OK
strultoa	Converts unsigned long to a string	stdlib.h	OK

OK : Re-entrant
NG : Not re-entrant

(6) String and Memory Functions

Function Name	Purpose	Header File	Re-entrant
memcpy	Copies a buffer for the specified number of characters	string.h	OK
memmove	Copies a buffer for the specified number of characters	string.h	OK
strcpy	Copies a string	string.h	OK
strncpy	Copies the specified number of characters from the start of a string	string.h	OK
strcat	Appends a string to a string	string.h	OK
strncat	Appends the specified number of characters of a string to a string	string.h	OK
memcmp	Compares the specified number of characters of two buffers	string.h	OK
strcmp	Compares two strings	string.h	OK
strncmp	Compares the specified number of characters of two strings	string.h	OK
memchr	Searches for the specified string in the specified number of characters of a buffer	string.h	OK
strchr	Searches for the specified character from within a string and returns the location of the first occurrence	string.h	OK
strrchr	Searches for the specified character from within a string and returns the location of the last occurrence	string.h	OK
strspn	Obtains the length from the start of a segment composed of only the characters included in the specified string within the string being searched	string.h	OK
strcspn	Obtains the length from the start of a segment composed of characters other than those included in the specified string within the string being searched	string.h	OK
strpbrk	Obtains the position of the first occurrence of any character in the specified string within the string being searched	string.h	OK
strstr	Obtains the position of the first occurrence of the specified string within the string being searched	string.h	OK
strtok	Decomposing character string into a string consisting of characters other than delimiters.	string.h	NG
memset	Initializes the specified number of characters of a buffer with the specified character	string.h	OK
strerror	Returns a pointer to the area that stores the error message string which corresponds to the specified error number	string.h	OK
strlen	Obtains the length of a string	string.h	OK
strcoll	Compares two strings based on region specific information	string.h	OK
strxfrm	Transforms a string based on region specific information	string.h	OK

OK : Re-entrant
NG : Not re-entrant

(7) Mathematical Functions

Function Name	Purpose	Header File	Re-entrant
acos	Finds acos	math.h	NG
asin	Finds asin	math.h	NG
atan	Finds atan	math.h	NG

Function Name	Purpose	Header File	Re-entrant
atan2	Finds atan2	math.h	NG
cos	Finds cos	math.h	NG
sin	Finds sin	math.h	NG
tan	Finds tan	math.h	NG
cosh	Finds cosh	math.h	NG
sinh	Finds sinh	math.h	NG
tanh	Finds tanh	math.h	NG
exp	Finds the eNGponential function	math.h	NG
frexp	Finds mantissa and eNGponent part	math.h	NG
ldexp	Finds NG * 2 ^ eNGp	math.h	NG
log	Finds the natural logarithm	math.h	NG
log10	Finds the base 10 logarithm	math.h	NG
modf	Finds the decimal and integer parts	math.h	NG
pow	Finds yth power of NG	math.h	NG
sqrt	Finds the square root	math.h	NG
ceil	Finds the smallest integer not smaller than NG	math.h	NG
fabs	Finds the absolute value of floating point number NG	math.h	NG
floor	Finds the largest integer not larger than NG	math.h	NG
fmod	Finds the remainder of NG/y	math.h	NG
matherr	Obtains the eNGception processing for the library handling floating point numbers	math.h	NG
acosf	Finds acos	math.h	NG
asinf	Finds asin	math.h	NG
atanf	Finds atan	math.h	NG
atan2f	Finds atan of y/NG	math.h	NG
cosf	Finds cos	math.h	NG
sinf	Finds sin	math.h	NG
tanf	Finds tan	math.h	NG
coshf	Finds cosh	math.h	NG
sinhf	Finds sinh	math.h	NG
tanhf	Finds tanh	math.h	NG
expf	Finds the eNGponential function	math.h	NG
frexpf	Finds mantissa and eNGponent part	math.h	NG
ldexpf	Finds NG * 2 ^ eNGp	math.h	NG
logf	Finds the natural logarithm	math.h	NG
log10f	Finds the base 10 logarithm	math.h	NG
modff	Finds the decimal and integer parts	math.h	NG
powf	Finds yth power of NG	math.h	NG
sqrtf	Finds the square root	math.h	NG
ceilf	Finds the smallest integer not smaller than NG	math.h	NG

Function Name	Purpose	Header File	Re-entrant
fabsf	Finds the absolute value of floating point number NG	math.h	NG
floorf	Finds the largest integer not larger than NG	math.h	NG
fmodf	Finds the remainder of NG/y	math.h	NG

NG : Not re-entrant

(8) Diagnostic Function

Function Name	Purpose	Header File	Re-entrant
__assertfail	Supports the assert macro	assert.h	OK

OK : Re-entrant

6.1.2 Runtime library

This section shows the functions included in the runtime library.

These operation instructions are called in a format with @@ attached to the beginning of the function name. However, cstart, cstarte, cprep, and cdisp are called in a format with _@ attached to the beginning of the function name.

In addition, operations that do not appear in the tables below have no library support. The compiler performs inline expansion.

long addition/subtraction, and/or/xor, and shift may also undergo inline expansion.

(1) Increment

Function Name	Purpose
lsinc	Increments signed long
luinc	Increments unsigned long
finc	Increments float

(2) Decrement

Function Name	Purpose
lsdec	Decrements signed long
ludec	Decrements unsigned long
fdec	Decrements float

(3) Sign reverse

Function Name	Purpose
lsrev	Reverses the sign of signed long
lurev	Reverses the sign of unsigned long
frev	Reverses the sign of float

(4) 1's complement

Function Name	Purpose
lscom	Obtains 1's complement of signed long
lucom	Obtains 1's complement of unsigned long

(5) Logical negation

Function Name	Purpose
lsnot	Negates signed long
lunot	Negates unsigned long

(6) Multiplication

Function Name	Purpose
csmul	Performs multiplication between signed char data
cumul	Performs multiplication between unsigned char data
iumul	Performs multiplication between unsigned int data
lsmul	Performs multiplication between signed long data
lumul	Performs multiplication between unsigned long data
fmul	Performs multiplication between float data

(7) Division

Function Name	Purpose
csdiv	Performs division between signed char data
cudiv	Performs division between unsigned char data
isdiv	Performs division between signed int data
iudiv	Performs division between unsigned int data
lsdiv	Performs division between signed long data
ludiv	Performs division between unsigned long data
fdiv	Performs division between float data

(8) Remainder arithmetic

Function Name	Purpose
csrem	Performs remainder arithmetic between signed char data
curem	Performs remainder arithmetic between unsigned char data
isrem	Performs remainder arithmetic between signed int data
iurem	Performs remainder arithmetic between unsigned int data
lsrem	Performs remainder arithmetic between signed long data
lurem	Performs remainder arithmetic between unsigned long data

(9) Addition

Function Name	Purpose
lsadd	Performs addition between signed long data
luadd	Performs addition between unsigned long data
fadd	Performs addition between float data

(10) Subtraction

Function Name	Purpose
lssub	Performs subtraction between signed long data
lusub	Performs subtraction between unsigned long data
fsub	Performs subtraction between float data

(11) Left shift

Function Name	Purpose
lslsh	Performs left shift of signed long data
lulsh	Performs left shift of unsigned long data

(12) Right shift

Function Name	Purpose
lsrsh	Performs right shift of signed long data
lursh	Performs right shift of unsigned long data

(13) Compare

Function Name	Purpose
cscmp	Compares signed char data
iscmp	Compares signed int data
lscmp	Compares signed long data
lucmp	Compares unsigned long data
fcmp	Compares float data

(14) Bit AND

Function Name	Purpose
lsband	Performs AND operation between signed long data
luband	Performs AND operation between unsigned long data

(15) Bit OR

Function Name	Purpose
lsbor	Performs OR operation between signed long data
lubor	Performs OR operation between unsigned long data

(16) Bit XOR

Function Name	Purpose
lsbNGor	Performs XOR operation between signed long data
lubNGor	Performs XOR operation between unsigned long data

(17) Conversion from floating point

Function Name	Purpose
ftols	Converts from float to signed long
ftolu	Converts from float to unsigned long

(18) Conversion to floating point

Function Name	Purpose
lstof	Converts from signed long to float
lutof	Converts from unsigned long to float

(19) Conversion from bit

Function Name	Purpose
btol	Converts a bit to long

(20) Startup routine

Function Name	Purpose

cstart	Startup module
	- Acquires the area (4 * 32 bytes) to register functions with the ateNGit function and makes the beginning label name _@FNCTBL
	- Acquires the break area (32 bytes) and makes the beginning label name _@MEMTOP and the area's neNGt address label name _@MEMBTM
	- Defines the reset vector table's segment in the following manner and specifies the startup module's beginning address
	`@@VECT00 CSEG AT 0000H`
	`DW _@cstart`
	- Sets the mirror region
	- Sets the register bank to RB0
	- Sets the variable _errno, which inputs the error number, to 0
	- Sets the variable _@FNCENT, which inputs the number of functions registered with the ateNGit function, to 0
	- Initializes the break value and sets _@MEMTOP's address to the variable _@BRKADR

Function Name	Purpose
	- Set 1 as the initial value for the variable _@SEED, which is the source of pseudo random numbers for the rand function - Perform copy processing of initialized data and eNGecute 0 - Clear of eNGternal data without an initial value. - Calls the main function (user program) - Calls the eNGit function with the argument 0

(21) Function pre- and post-processing

Function Name	Purpose
cpre3	Performs pre-processing for functions (includes the saddr area for register variables)
cprep3	Performs pre-processing for functions (includes the saddr area for register variables)
cdis3e	Performs post-processing for functions (includes the saddr area for register variables)
cdisp3	Performs post-processing for functions (includes the saddr area for register variables)
cpre3e	Performs pre-processing for functions (includes the saddr area for register variables)
cdis3e	Performs post-processing for functions (includes the saddr area for register variables)
hdwinit	Performs initialization of peripheral devices (sfr) immediately after a CPU reset

(22) BCD-type conversion

Function Name	Purpose
bcdtob	Converts 1-byte bcd to 1-byte binary
btobcd	Converts 1-byte binary to 2-byte bcd
bcdtow	Converts 2-byte bcd to 2-byte binary
wtobcd	Converts 2-byte binary to 2-byte bcd
bbcd	Converts 1-byte binary to 2-byte bcd

(23) Auxiliary

Function Name	Purpose
divuw	divuw instruction compatibility
df1in	For replacing fiNGed instruction pattern
dn4in	For replacing fiNGed instruction pattern
dn4ip	For replacing fiNGed instruction pattern
df4in	For replacing fiNGed instruction pattern
df4ip	For replacing fiNGed instruction pattern
dn4ino	For replacing fiNGed instruction pattern
dn4ipo	For replacing fiNGed instruction pattern
df4ino	For replacing fiNGed instruction pattern
df4ipo	For replacing fiNGed instruction pattern
df1de	For replacing fiNGed instruction pattern
dn4de	For replacing fiNGed instruction pattern
dn4dp	For replacing fiNGed instruction pattern

Function Name	Purpose
df4de	For replacing fiNGed instruction pattern
df4dp	For replacing fiNGed instruction pattern
dn4deo	For replacing fiNGed instruction pattern
dn4dpo	For replacing fiNGed instruction pattern
df4deo	For replacing fiNGed instruction pattern
df4dpo	For replacing fiNGed instruction pattern
indao	For replacing fiNGed instruction pattern
ifdao	For replacing fiNGed instruction pattern
inado	For replacing fiNGed instruction pattern
ifado	For replacing fiNGed instruction pattern
lnd0	For replacing fiNGed instruction pattern
lfd0	For replacing fiNGed instruction pattern
ln0d	For replacing fiNGed instruction pattern
lf0d	For replacing fiNGed instruction pattern
lnd0o	For replacing fiNGed instruction pattern
lfd0o	For replacing fiNGed instruction pattern
ln0do	For replacing fiNGed instruction pattern
lf0do	For replacing fiNGed instruction pattern

6.2 Interface Between Functions

Library functions are used with function calls. Function calls are done with the call instruction. Arguments are passed on the stack, return values are passed by registers.

However, if possible, the first argument is also passed by registers.

6.2.1 Arguments

The function interface (passing arguments, storing return values) for the standard library is the same as that of regular functions.

For details see "3.4.2 Ordinary function call interface".

6.2.2 Return values

Return values are a minimum of 16 bits and are stored in 16-bit units from the low-order bits from register BC to DE. When returning a structure, the structure's starting address is stored in BC, DE.

For details see "3.4.1 Return values".

6.2.3 Saving registers used by separate libraries

Libraries that use HL save registers that use those to the stack.

Libraries that use the saddr area save the saddr area to use to the stack.

The work area used by libraries also use the stack area.

An example (for small model, medium model) of the procedure for passing arguments and return values is shown next.

The following show the function to call.

```
"long func ( int a, long b, char *c );"
```

(1) Push arguments on the stack (function call source)

The arguments are pushed onto the stack in the order c, b in the high-order 16 bits, b in the low-order 16 bits. a is passed in the AX register.

(2) Call func with the call instruction (function call source)

After b in the low-order 16 bits, the return address is pushed onto the stack, control moves to the function func.

(3) Save the registers to use in the function (function call target)

When using HL, HL is pushed onto the stack.

(4) Push the first argument passed by the register onto the stack (function call target)

(5) Perform processing for the function func, store the return value in a register (function call target)

The low-order 16 bits of the return value "long" are stored in BC, the high-order 16 bits are stored in DE.

(6) Restore the stored first argument (function call target)

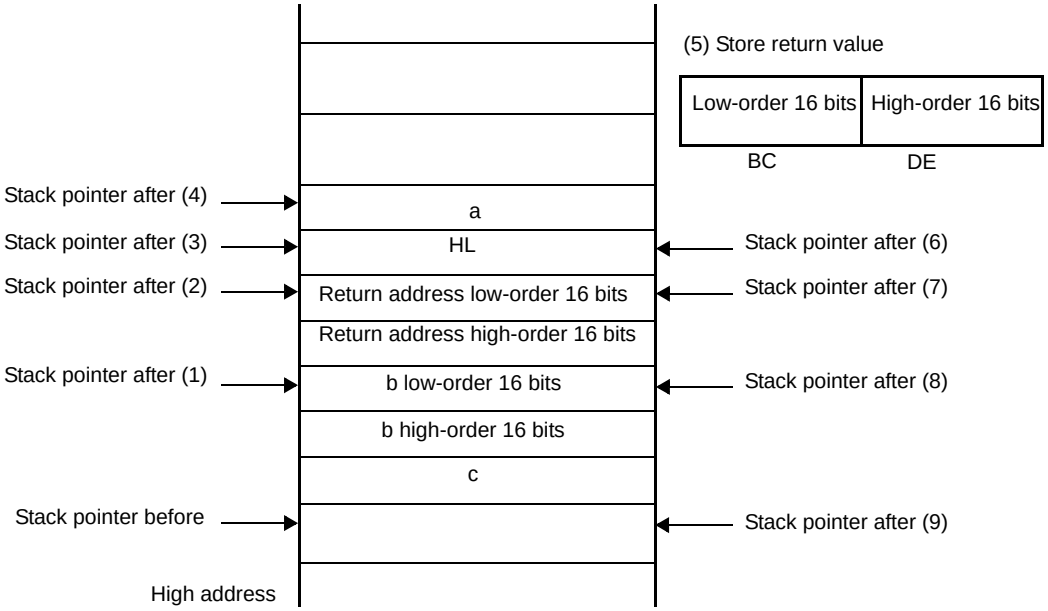
(7) Restore the saved registers (function call target)

(8) Return control to the calling function with the ret instruction (function call target)

(9) Clear the arguments off the stack (function call source)

The number of bytes (2-byte units) of the arguments is added to the stack pointer.
6 is added to the stack pointer.

Figure 6-1. Stack Area During Function Call



6.3 Header Files

There are 13 header files in 78K0R C compiler and they define and declare the standard library functions, type names, and macro names.

78K0R C compiler header files are shown next.

6.3.1 ctype.h

ctype.h defines character/string functions.

In ctype.h, the following functions are defined.

However, when compiler option -za (the option to turn off non-ANSI compliant functions and turn on a portion of ANSI compliant functions) is specified, _toupper and _tolower are not defined, in their place tolow and toup are defined. When -za is not specified, tolow and toup are not defined. The functions declared also vary depending on the options and specific model.

```
isalpha, isupper, islower, isdigit, isalnum, isxdigit, isspace, ispunct, isprint, isgraph,
iscntrl, isascii, toupper, tolower, toascii, _toupper/toup, _tolower/tolow
```

6.3.2 setjmp.h

setjmp.h defines program control functions.

In setjmp.h, the following functions are defined. In addition, the functions declared also vary depending on the options and specific model.

```
setjmp, longjmp
```

In setjmp.h, the following object is declared.

- Declaration of the int array type "jmp_buf"

```
typedef int     jmp_buf[12]
```

6.3.3 stdarg.h

stdarg.h defines special functions.

In stdarg.h, the following functions are defined.

```
va_start, va_starttop, va_arg, va_end
```

In stdarg.h, the following object is defined.

- Declaration of the pointer type "va_list" to char

```
typedef char    *va_list  ;
```

6.3.4 stdio.h

stdio.h defines input/output functions. In stdio.h, the following functions are defined.

However, the functions declared vary depending on the options and specific model

```
sprintf, sscanf, printf, scanf, vprintf, vsprintf, getchar, gets, putchar, puts, __putc
```

The following macro name is declared.

```
#define  EOF     ( -1 )
```

6.3.5 stdlib.h

stdlib.h defines character/string functions, memory functions, program control, math functions, and special functions. In stdlib.h, the following functions are defined.

However, when compiler option -za (the option to turn off non-ANSI compliant functions and turn on a portion of ANSI compliant functions) is specified, brk, sbrk, itoa, ltoa, and ultoa are not defined, in their place strbrk, strsbrk, stritoa, strltoa, and strultoa are defined. When -za is not specified, these functions are not defined.

```
atoi, atol, strtol, strtoul, calloc, free, malloc, realloc, abort, atexit, exit, abs, labs,
div, ldiv, brk, sbrk, atof, strtod, itoa, ltoa, ultoa, rand, srand, bsearch, qsort, strbrk,
strsbrk, stritoa, strltoa, strultoa
```

In stdlib.h, the following objects are declared.

- Declaration of the structure type "div_t" with int members "quot" and "rem".

```
typedef struct {
        int     quot ;
        int     rem ;
} div_t ;
```

- Definition of the macro name "RAND_MAX"

```
#define RAND_MAX        32767
```

- Macro name declarations

```
#define EXIT_SUCCESS    0
#define EXIT_FAILURE    1
```

6.3.6 string.h

string.h defines character/string functions, memory functions, and special functions.
In string.h, the following functions are defined.
However, the functions declared vary depending on the options and specific model.

```
memcpy, memmove, strcpy, strncpy, strcat, strncat, memcmp, strcmp, strncmp, memchr, strchr,
strrchr, strspn, strcspn, strpbrk, strstr, strtok, memset, strerror, strlen, strcoll,
strxfrm
```

6.3.7 error.h

error.h includes errno.h.

6.3.8 errno.h

The following objects are declared or defined

- Definition of the macro names "EDOM", "ERANGE", "ENOMEM"

```
#define EDOM    1
#define ERANGE  2
#define ENOMEM  3
```

- eclaration of the external variable "errno" of the volatile int type

```
extern volatile int errno ;
```

6.3.9 limits.h

In limits.h, the following macro names are defined.

```
#define CHAR_BIT        8
#define CHAR_MAX        +127
#define CHAR_MIN        -128
#define INT_MAX         +32767
#define INT_MIN         -32768
#define LONG_MAX        +2147483647
#define LONG_MIN        -2147483648

#define SCHAR_MAX       +127
#define SCHAR_MIN       -128
#define SHRT_MAX        +32767
#define SHRT_MIN        -32768
#define UCHAR_MAX       255U
#define UINT_MAX        65535U
#define ULONG_MAX       4294967295U
#define USHRT_MAX       65535U

#define SINT_MAX        +32767
#define SINT_MIN        -32768
#define SSHRT_MAX       +32767
#define SSHRT_MIN       -32768
```

However, when the -qu option is specified, which considers an unmodified char as an unsigned char, CHAR_MAX and CHAR_MIN are declared in the following manner by the macro __CHAR_UNSIGNED__ declared by the compiler.

```
#define CHAR_MAX        ( 255U )
#define CHAR_MIN        ( 0 )
```

6.3.10 stddef.h

In stddef.h, the following objects are declared or defined.

- Declaration of the int type "ptrdiff_t

```
typedef int     ptrdiff_t ;
```

- Declaration of the unsigned int type "size_t"

```
typedef unsigned int    size_t ;
```

- Definition of the macro name "NULL"

```
#define NULL    ( void * ) 0 ;
```

- Definition of the macro name "offsetof"

```
#define offsetof ( type, member )  ( (size_t) & (((type*)0) -> member) )
```

Remark offsetof (*type*, *member-designator*)
Expands to a general integer constant expression of type size_t, and that value is the offset value in bytes from the start of the structure (specified by the type) to the structure member (specified by the member designator).
When the member specifier has been declared as static type t;, the result of evaluating expression& (t. member specifier) must be an address constant. When the specified member is a bit field, there is no guarantee of the operation.

6.3.11 math.h

In math.h, the following functions are defined

```
acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10,
modf, pow, sqrt, ceil, fabs, floor, fmod, matherr, acosf, asinf, atanf, atan2f, cosf, sinf,
tanf, coshf, sinhf, tanhf, expf, frexpf, ldexpf, logf, log10f, modff, powf, sqrtf, ceilf,
fabsf, floorf, fmodf
```

The following objects are defined.

- Definition of the macro name "HUGE_VAL"

```
#define HUGE_VAL        DBL_MAX
```

6.3.12 float.h

In float.h, the following objects are defined.

The macros defined are split according to the macro __DOUBLE_IS_32BITS__ which is declared by the compiler when the size of the double type is 32 bits.

```
#ifndef _FLOAT_H

#define FLT_ROUNDS      1
#define FLT_RADIX       2

#ifdef  __DOUBLE_IS_32BITS__
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    24
#define LDBL_MANT_DIG   24

#define FLT_DIG         6
#define DBL_DIG         6
#define LDBL_DIG        6

#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -125
#define LDBL_MIN_EXP    -125

#define FLT_MIN_10_EXP  -37
#define DBL_MIN_10_EXP  -37
#define LDBL_MIN_10_EXP -37

#define FLT_MAX_EXP     +128
#define DBL_MAX_EXP     +128
#define LDBL_MAX_EXP    +128

#define FLT_MAX_10_EXP  +38
#define DBL_MAX_10_EXP  +38
#define LDBL_MAX_10_EXP +38

#define FLT_MAX         3.40282347E+38F
#define DBL_MAX         3.40282347E+38F
#define LDBL_MAX        3.40282347E+38F

#define FLT_EPSILON     1.19209290E-07F
#define DBL_EPSILON     1.19209290E-07F
#define LDBL_EPSILON    1.19209290E-07F

#define FLT_MIN         1.17549435E-38F
#define DBL_MIN         1.17549435E-38F
#define LDBL_MIN        1.17549435E-38F

#else   /* __DOUBLE_IS_32BITS__ */
```

```
#define FLT_MANT_DIG    24
#define DBL_MANT_DIG    53
#define LDBL_MANT_DIG   53

#define FLT_DIG         6
#define DBL_DIG         15
#define LDBL_DIG        15

#define FLT_MIN_EXP     -125
#define DBL_MIN_EXP     -1021
#define LDBL_MIN_EXP    -1021

#define FLT_MIN_10_EXP  -37
#define DBL_MIN_10_EXP  -307
#define LDBL_MIN_10_EXP -307

#define FLT_MAX_EXP     +128
#define DBL_MAX_EXP     +1024
#define LDBL_MAX_EXP    +1024

#define FLT_MAX_10_EXP  +38
#define DBL_MAX_10_EXP  +308
#define LDBL_MAX_10_EXP +308

#define FLT_MAX         3.40282347E+38F
#define DBL_MAX         1.7976931348623157E+308
#define LDBL_MAX        1.7976931348623157E+308

#define FLT_EPSILON     1.19209290E-07F
#define DBL_EPSILON     2.2204460492503131E-016
#define LDBL_EPSILON    2.2204460492503131E-016

#define FLT_MIN         1.17549435E-38F
#define DBL_MIN         2.225073858507201E-308
#define LDBL_MIN        2.225073858507201E-308
#endif  /* __DOUBLE_IS_32BITS__ */

#define _FLOAT_H
#endif  /* !_FLOAT_H */
```

6.3.13 assert.h

In assert.h, the following functions are defined.

```
__assertfail
```

In assert.h, the following objects are defined.

```
#ifdef  NDEBUG
#define assert ( p )    ( ( void ) 0 )
#else
extern  int     __assertfail ( char *__msg, char *__cond, char *__file, int__line );
#define assert ( p )    ( ( p ) ? ( void ) 0 : ( void ) __assertfail (
                            "Assertion failed : %s, file %s, line %d\n",
                        #p, __FILE__, __LINE__ ) )
#endif  /* NDEBUG */
```

However, the assert.h header file references one more macro, NDEBUG, which is not defined in the assert.h header file. If NDEBUG is defined as a macro when assert.h is included in the source file, the assert macro is simply defined as shown next, and __assertfail is also not defined.

```
#define assert ( p )    ( ( void ) 0 )
```

6.4 Re-entrant

Re-entrant is a state where it is possible for a function called by a program to be successively called by another program.

The 78K0R C compiler standard library takes re-entrantability into consideration and does not use static areas. Therefore, the data in the memory area used by the function is not damaged by a call from another program.

However, be careful as the following functions are not re-entrant.

- Functions that cannot be made re-entrant

```
setjmp, longjmp, atexit, exit
```

- Functions that use the area acquired by the startup routine

```
div, ldiv, brk, sbrk, rand, srand, strtok
```

- Functions that handle floating point numbers

```
sprintf, sscanf, printf, scanf, vprintf, vsprintf[Note]
atof, strtod, all math functions
```

Note Of sprintf, sscanf, printf, scanf, vprintf, and vsprintf, functions that do not support floating point are re-entrant.

6.5 Use of Arguments/Return Values Suitable for Standard Library

Functions that specify pointers to arguments/return values of the standard library are linked to the appropriate library according to the memory model.

To handle pointers that are not of the default memory model, it is possible to link to the appropriate library for that pointer by calling the functions with the standard function names below.

<function name>_n: : always handles the pointer as near

<function nam>_f: : always handles the pointer as far

For example, when the small model is selected, the far pointers can be specified as an argument for the strcmp function.

An example is shown below.

```
#include <string.h>

__far char * sf1;
__far char * sf2;

void main ( void ) {
          :
        r = strcmp_f ( sf1, sf2 );
          :
}
```

Cautions are shown below.

- When the small model and medium model are specified, pointer arguments for the input/output functions sprintf/printf/vprintf/vsprintf/sscanf/scanf, which handle variable arguments, are handled as near pointers. Function pointers cannot be used.
  When using function pointers, or when using far pointers, use printf_f and it will be necessary to cast all of the variable argument pointers to far pointers.
  When the large model is specified, pointer arguments for the input/output functions sprintf/printf/vprintf/vsprintf/sscanf/scanf, which handle variable arguments, are handled as far pointers.
- When the small model and medium model are specified, pointer arguments for the special functions va_start/va_starttop/va_arg/va_end, which handle variable arguments, are handled as near pointers. Function pointers cannot be used.

6.6 Character/String Functions

The following functions are available as character/string functions.

Function Name	Purpose
isalpha	Judges if a character is an alphabetic character (A to Z, a to z)
isupper	Judges if a character is an uppercase alphabetic character (A to Z)
islower	Judges if a character is an lowercase alphabetic character (a to z)
isdigit	Judges if a character is a numeric (0 to 9)
isalnum	Judges if a character is an alphanumeric character (0 to 9, A to Z, a to z)
isxdigit	Judges if a character is a hexadecimal numbers (0 to 9, A to F, a to f)
isspace	Judges if a character is a whitespace character (whitespace, tab, carriage return, line feed, vertical, tab, form feed)
ispunct	Judges if a character is a printable character other than a whitespace character or alphanumeric character
isprint	Judges if a character is a printable character
isgraph	Judges if a character is a printable character other than whitespace
iscntrl	Judges if a character is a control character
isascii	Judges if a character is an ASCII code
toupper	Converts a lowercase alphabetic character to uppercase
tolower	Converts an uppercase alphabetic character to lowercase
toascii	Converts the input to an ASCII code
_toupper	Subtracts "a" from the input character and adds "A"
toup	
_tolower	Subtracts "A" from the input character and adds "a"
tolow	

isalpha

Judges if *c* is an alphabetic character (A to Z, a to z)

[Syntax]

```
#include <ctype.h>
int         isalpha ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in alphabetic character (A to Z or a to z) : 1 If character *c* is not included in alphabetic character (A to Z or a to z) : 0

[Description]

- If character *c* is included in alphabetic character (A to Z or a to z), 1 is returned.
  In other cases, 0 is returned.

isupper

Judges if *c* is an uppercase alphabetic character (A to Z)

[Syntax]

```
#include <ctype.h>
int     isupper ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the uppercase letters (A to Z) : 1 If character *c* is not included in the uppercase letters (A to Z) : 0

[Description]

- If character *c* is included in uppercase letters character (A to Z), 1 is returned.
  In other cases, 0 is returned.

islower

Judges if *c* is an lowercase alphabetic character (a to z)

[Syntax]

```
#include <ctype.h>
int         islower ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the lowercase letters (a to z) : 1 If character *c* is not included in the lowercase letters (a to z) : 0

[Description]

- If character *c* is included in the lowercase letters (a to z) , 1 is returned.
  In other cases, 0 is returned.

isdigit

Judges if *c* is a numeric (0 to 9)

[Syntax]

```
#include <ctype.h>
int     isdigit ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the numeric characters (0 to 9) : 1 If character *c* is not included in the numeric characters (0 to 9) : 0

[Description]

- If character *c* is included in the numeric characters (0 to 9) , 1 is returned.
  In other cases, 0 is returned.

isalnum

Judges if *c* is an alphanumeric character (0 to 9, A to Z, a to z)

[Syntax]

```
#include <ctype.h>
int         isalnum ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the alphanumeric characters (0 to 9 and A to Z or a to z) : 1 If character *c* is not included in the alphanumeric characters (0 to 9 and A to Z or a to z) : 0

[Description]

- If character *c* is included in the alphanumeric characters (0 to 9 and A to Z or a to z) , 1 is returned.
  In other cases, 0 is returned.

isxdigit

Judges if *c* is a hexadecimal numbers (0 to 9, A to F, a to f)

[Syntax]

```
#include <ctype.h>
int         isxdigit ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the hexadecimal numbers (0 to 9 and A to F or a to f) : 1 If character *c* is not included in the hexadecimal numbers (0 to 9 and A to F or a to f) : 0

[Description]

- If character *c* is included in the hexadecimal numbers (0 to 9 and A to F or a to f), 1 is returned.
  In other cases, 0 is returned.

isspace

Judges if *c* is a whitespace character (space, tab, carriage return, line feed, vertical, tab, form feed)

[Syntax]

```
#include <ctype.h>
int         isspace ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the whitespace characters : 1 If character *c* is not included in the whitespace characters : 0

[Description]

- If character *c* is included in the whitespace characters (space, tab, carriage return, line feed, vertical, tab, form feed), 1 is returned.
  In other cases, 0 is returned.

ispunct

Judges if *c* is a printable character other than a whitespace character or alphanumeric character

[Syntax]

```
#include <ctype.h>
int         ispunct ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the printable characters except whitespace and alphanumeric characters : 1 If character *c* is not included in the printable characters except whitespace and alphanumeric characters : 0

[Description]

- If character *c* is included in the printable characters except whitespace and alphanumeric characters, 1 is returned.
  In other cases, 0 is returned.

isprint

Judges if *c* is a character is a printable character

[Syntax]

```
#include <ctype.h>
int         isprint ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the printable characters : 1 If character *c* is not included in the printable characters : 0

[Description]

- If character *c* is included in the printable characters, 1 is returned.
  In other cases, 0 is returned.

isgraph

Judges if *c* is a printable character other than whitespace

[Syntax]

```
#include <ctype.h>
int     isgraph ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the printable nonblank characters : 1 If character *c* is not included in the printable nonblank characters : 0

[Description]

- If character *c* is included in the printable nonblank character, 1 is returned.
  In other cases, 0 is returned.

iscntrl

Judges if *c* is a control character

[Syntax]

```
#include <ctype.h>
int         iscntrl ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the control characters : 1 If character *c* is not included in the control characters : 0

[Description]

- If character *c* is included in the control characters , 1 is returned.
  In other cases, 0 is returned.

isascii

Judges if *c* is an ASCII code

[Syntax]

```
#include <ctype.h>
int         isascii( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be judged	If character *c* is included in the ASCII code set : 1 If character *c* is not included in the ASCII code set : 0

[Description]

- If character *c* is included in the ASCII code set , 1 is returned.
  In other cases, 0 is returned.

toupper

Converts a lowercase alphabetic character to uppercase

[Syntax]

```
#include <ctype.h>
int     toupper ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	If *c* is a convertible character : uppercase equivalent of *c* If not convertible : c

[Description]

- The toupper function checks to see if the argument is a lowercase letter and if so converts the letter to its uppercase equivalent.

tolower

Converts an uppercase alphabetic character to lowercase

[Syntax]

```
#include <ctype.h>
int     tolower ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	If *c* is a convertible character : lowercase equivalent of *c* If not convertible : c

[Description]

- The tolower function checks to see if the argument is a uppercase letter and if so converts the letter to its lowercase equivalent.

toascii

Converts the input to an ASCII code

[Syntax]

```
#include <ctype.h>
int         toascii ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	Value obtained by converting the bits outside the ASCII code range of "*c*" to 0.

[Description]

- The toascii function converts the bits (bits 7 to 15) of "*c*" outside the ASCII code range of "*c*" (bits 0 to 6) to "0" and returns the converted bit value.

_toupper

Subtracts "a" from "*c*" and adds "A" to the result
(_toupper is exactly the same as toup)

Remark a: Lowercase; A: Uppercase

[Syntax]

```
#include <ctype.h>
int     _toupper ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	Value obtained by adding "A" to the result of subtraction "*c*" - "a"

Remark a: Lowercase; A: Uppercase

[Description]

- The _toupper function is similar to toup except that it does not test to see if the argument is a lowercase letter.

toup

Subtracts "a" from "*c*" and adds "A" to the result
(_toupper is exactly the same as toup)

Remark a: Lowercase; A: Uppercase

[Syntax]

```
#include <ctype.h>
int         toup ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	Value obtained by adding "A" to the result of subtraction "*c*" - "a"

Remark a: Lowercase; A: Uppercase

[Description]

- The toup function is similar to _toupper except that it tests to see if the argument is a lowercase letter.

_tolower

Subtracts "A" from "*c*" and adds "a" to the result
(_tolower is exactly the same as the tolow)

Remark a: Lowercase; A: Uppercase

[Syntax]

```
#include <ctype.h>
int     _tolower ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	Value obtained by adding "a" to the result of subtraction "*c*" - "A"

Remark a: Lowercase; A: Uppercase

[Description]

- The _tolower function is similar to tolow, except it does not test to see if the argument is an uppercase letter.

tolow

Subtracts "A" from "*c*" and adds "a" to the result
(_tolower is exactly the same as the tolow)

Remark a: Lowercase; A: Uppercase

[Syntax]

```
#include <ctype.h>
int         tolow ( int c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be converted	Value obtained by adding "a" to the result of subtraction "*c*" - "A"

Remark a: Lowercase; A: Uppercase

[Description]

- The tolow function is similar to _tolower, except it tests to see if the argument is an uppercase letter.

6.7 Program Control Functions

The following program control functions are available.

Function Name	Purpose
setjmp	Saves the environment at the time of the call
longjmp	Restores the environment saved with setjmp

setjmp

Saves the environment at the time of the call

[Syntax]

#include <setjmp.h>
int setjmp (jmp_buf *env*) ;

[Argument(s)/Return value]

Argument	Return Value
env : Array to which environment information is to be saved	If called directly : 0 If returning from the corresponding longjmp : Value given by "*val*" or 1 if "*val* " is 0.

[Description]

- The setjmp, when called directly, saves saddr area, SP, and the return address of the function that are used as HL register or register variables to *env* and returns 0.

longjmp

Restores the environment saved with setjmp

[Syntax]

#include <setjmp.h>
void longjmp (jmp_buf *env* , int *val*) ;

[Argument(s)/Return value]

Argument	Return Value
env : Array to which environment information was saved by setjmp *val* : Return value to setjmp	longjmp will not return because program execution resumes at statement next to setjmp that saved environment to "*env*".

[Description]

- The longjmp restores the saved environment to *env* (HL register, saddr area and SP that are used as register variables). Program execution continues as if the corresponding setjmp returns *val* (however, if *val* is 0, 1 is returned).

6.8 Special Functions

The following special functions are available.

Function Name	Purpose
va_start	Settings for processing variable arguments
va_starttop	Settings for processing variable arguments
va_arg	Processes variable arguments
va_end	Indicates the end of processing variable arguments

va_start

Settings for processing variable arguments (macro)

[Syntax]

#include <stdarg.h>
void va_start (va_list *ap* , *parmN*) ;

Remark va_list is defined as typedef by stdarg.h.

[Argument(s)/Return value]

Argument	Return Value
ap : Variable to be initialized so as to be used in va_arg and va_end *parmN* : The argument before variable argument	None

[Description]

- In the va_start macro, its argument *ap* must be a va_list type (char* type) object.
- A pointer to the next argument of *parmN* is stored in *ap*.
- *parmN* is the name of the last (right-most) parameter specified in the function's prototype.
- If *parmN* has the register storage class, proper operation of this function is not guaranteed.
- If *parmN* is the first argument, this function may not operate normally (use va_starttop instead).

va_starttop

Settings for processing variable arguments (macro)

[Syntax]

```
#include <stdarg.h>
void  va_starttop ( va_list ap , parmN ) ;
```

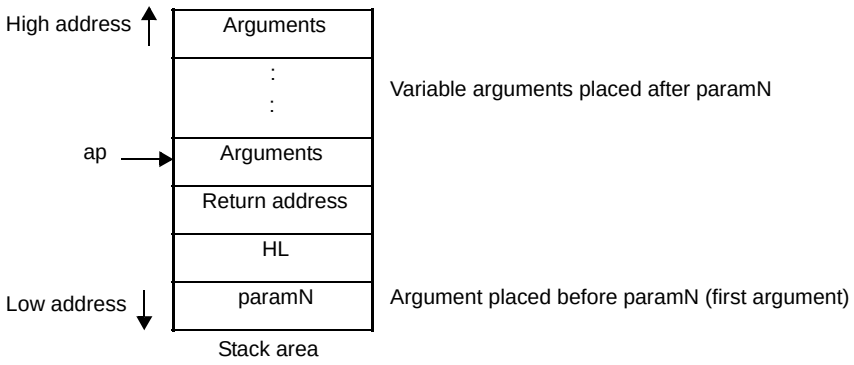
Remark va_list is defined as typedef by stdarg.h.

[Argument(s)/Return value]

Argument	Return Value
ap : Variable to be initialized so as to be used in va_arg and va_end *parmN* : The argument before variable argument	None

[Description]

- *ap* must be a va_list type object.
- A pointer to the next argument of *parmN* is stored in *ap*.
- *parmN* is the name of the right-most and first parameter specified in the function's prototype.
- If *parmN* has the register storage class, this function may not operate normally.
- If *parmN* is an argument other than the first argument, this function may not operate normally.



va_arg

Processes variable arguments (macro)

[Syntax]

#include <stdarg.h>
type va_arg (va_list *ap* , type) ;

Remark va_list is defined as typedef by stdarg.h.

[Argument(s)/Return value]

Argument	Return Value
ap : Variable to process an argument list type : Type to point the relevant place of variable argument (type is a type of variable length; for example, int type if described as va_arg (va_list *ap*, int) or long type if described as va_arg (va_list *ap*, long))	Normal case : Value in the relevant place of variable argument If *ap* is a null pointer : 0

[Description]

- In the va_arg macro, its argument *ap* must be the same as the va_list type object initialized with va_start (no guarantee for the other normal operation).
- va_arg returns value in the relevant place of variable arguments as a type of type.
  The relevant place is the first of variable arguments immediately after va_start and next proceeded in each va_arg.
- If the argument pointer *ap* is a null pointer, the va_arg returns 0 (of type type).
- With the 78K0R C compiler, when specifying a pointer as an argument list, the near data pointers (2-byte length) must be specified when the medium model is used, and the far data pointers (4-byte length) must be specified when the large model is used.
  The function pointer length is fixed to 4 bytes in both models, but the pointer length in each model must be specified as a 2- or 4-byte length when specifying the pointer as an argument list.

va_end

Indicates the end of processing variable arguments (macro)

[Syntax]

```
#include <stdarg.h>
void  va_end ( va_list ap ) ;
```

Remark va_list is defined as typedef by stdarg.h.

[Argument(s)/Return value]

Argument	Return Value
ap : Variable to process the variable number of arguments	None

[Description]

- The va_end macro sets a null pointer in the argument pointer *ap* to inform the macro processor that all the parameters in the variable argument list have been processed.

6.9 Input and Output Functions

The following input and output functions are available.

Function Name	Purpose
sprintf	Writes data to a string according to a format
sscanf	Reads data from the input string according to a format
printf	Outputs data to SFR according to a format
scanf	Reads data from SFR according to a format
vprintf	Outputs data to SFR according to a format
vsprintf	Writes data to a string according to a format
getchar	Reads one character from SFR
gets	Reads a string
putchar	Outputs one character to SFR
puts	Outputs a string
__putc	Outputs one character to opaque

sprintf

Writes data to a string according to a format

[Syntax]

#include <stdio.h>
int sprintf (char *s , const char *format , ...) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to the string into which the output is to be written *format* : Pointer to the string which indicates format commands ... : Zero or more arguments to be converted	Number of characters written in *s* (Terminating null character is not counted.)

[Description]

- If there are fewer actual arguments than the formats, the proper operation is not guaranteed. In the case that the formats are run out despite the actual arguments still remain, the excess actual arguments are only evaluated and ignored.
- sprintf converts zero or more arguments that follow *format* according to the format command specified by *format* and writes (copies) them into the string *s*.
- Zero or more format commands may be used. Ordinary characters (other than format commands that begin with a % character) are output as is to the string *s*. Each format command takes zero or more arguments that follow *format* and outputs them to the string *s*.
- Each format command begins with a % character and is followed by these:

(1) Zero or more flags (to be explained later) that modify the meaning of the format command

(2) Optional decimal integer which specify a minimum field width

If the output width after the conversion is less than this minimum field width, this specifier pads the output with blanks of zeros on its left. (If the left-justifying flag "-" (minus) sign follows %, zeros are padded out to the right of the output.) The default padding is done with spaces. If the output is to be padded with 0s, place a 0 before the field width specifier. If the number or string is greater than the minimum field width, it will be printed in full even by overrunning the minimum.

- Optional precision (number of decimal places) specification (.integer)
  With d, i, o, u, x, and X type specifiers, the minimum number of digits is specified.
  With s type specifier, the maximum number of characters (maximum field width) is specified.
  The number of digits to be output following the decimal point is specified for e, E, and f conversions. The number of maximum effective digits is specified for g and G conversions.
  This precision specification must be made in the form of (.integers). If the integer part is omitted, 0 is assumed to have been specified.
  The amount of padding resulting from this precision specification takes precedence over the padding by the field width specification.
- Optional h, l and L modifiers
  The h modifier instructs the sprintf function to perform the d, i, o, u, x, or X type conversion that follows this mod-

ifier on short int or unsigned short int type. The h modifier instructs the sprintf function to perform the n type conversion that follows this modifier on a pointer to short int type.

The l modifier instructs the sprintf function to perform the d, i, o, u, x, or X type conversion that follows this modifier on long int or unsigned long int type. The h modifier instructs the sprintf function to perform the n type conversion that follows this modifier on a pointer to long int type.

For other type specifiers, the h, l or L modifier is ignored.

- Character that specifies the conversion (to be explained later)

In the minimum field width or precision (number of decimal places) specification, * may be used in place of an integer string. In this case, the integer value will be given by the int argument (before the argument to be converted).

Any negative field width resulting from this will be interpreted as a positive field that follows the - (minus) flag.

All negative precision will be ignored.

The following flags are used to modify a format command:

Flag	Contents
-	The result of a conversion is left-justified within the field.
+	The result of a signed conversion always begins with a + or - sign.
space	If the result of a signed conversion has no sign, space is prefixed to the output. If the + (plus) flag and space flag are specified at the same time, the space flag will be ignored.
#	The result is converted in the "assignment form". In the o type conversion, precision is increased so that the first digit becomes 0. In the x or X type conversion, 0x or 0X is prefixed to a nonzero result. In the e, E, and f type conversions, a decimal point is forcibly inserted to all the output values (in the default without #, a decimal point is displayed only when there is a value to follow). In the g and G type conversions, a decimal point is forcibly inserted to all the output values, and truncation of 0 to follow will not be allowed (in the default without #, a decimal point is displayed only when there is a value to follow. The 0 to follow will be truncated). In all the other conversions, the # flag is ignored.
0	The result is left padded with zeros instead of spaces. The 0 flag is ignored when it is specified together with the "-" flag. The 0 flag is ignored in d, i, o, u, x, and X conversions when the precision is specified.

The format codes for output conversion specifications are as follows:

Format Code	Contents
d, i	Converts int argument to signed decimal format.
o	Converts int argument to signed decimal format.
u	Converts int argument to unsigned octal format.
x	Converts int argument to unsigned hexadecimal format (with lowercase letters abcdef).
X	Converts int argument to unsigned hexadecimal format (with uppercase letters ABCDEF).

With d, i, o, u, x and X type specifiers, the minimum number of digits (minimum field width) of the result is specified. If the output is shorter than the minimum field width, it is padded with zeros.

If no precision is specified, 1 is assumed to have been specified.

Nothing will appear if 0 is converted with 0 precision.

Precision Code	Contents
f	Converts double argument as a signed value with [-] *dddd*.*dddd* format. *dddd* is one or more decimal number(s). The number of digits before the decimal point is determined by the absolute value of the number, and the number of digits after the decimal point is determined by the required precision. When the precision is omitted, it is interpreted as 6.
e	Converts double argument as a signed value with [-] *d*.*dddd* e [sign] *ddd* format. d is 1 decimal number, and *dddd* is one or more decimal number(s). *ddd* is exactly a 3- digit decimal number, and the sign is + or -. When the precision is omitted, it is interpreted as 6.
E	The same format as that of e except E is added instead of e before the exponent.
g	Uses whichever shorter method of f or e format when converting double argument based on the specified precision. e format is used only when the exponent of the value is smaller than -4 or larger than the specified number by precision. The following 0 are truncated, and the decimal point is displayed only when one or more numbers follow.
G	The same format as that of g except E is added instead of e before the exponent.
c	Converts int argument to unsigned char and writes the result as a single character.
s	The associated argument is a pointer to a string of characters and the characters in the string are written up to the terminating null character (but not included in the output). If precision is specified, the characters exceeding the maximum field width will be truncated off the end. When the precision is not specified or larger than the array, the array must include a null character.
p	The associated argument is a pointer to void and the pointer value is displayed in unsigned hexadecimal 4 digits (with 0s prefixed to less than a 4-digit pointer value). The large model is displayed in unsigned hexadecimal 8 digits (with 0 padded in dominance 2-digits and 0s prefixed to less than a 6-digit pointer value). The precision specification if any will be ignored.
n	The associated argument is an integer pointer into which the number of characters written thus far in the string "s" is placed. No conversion is performed.
%	Prints a % sign. The associated argument is not converted (but the flag and minimum field width specifications are effective).

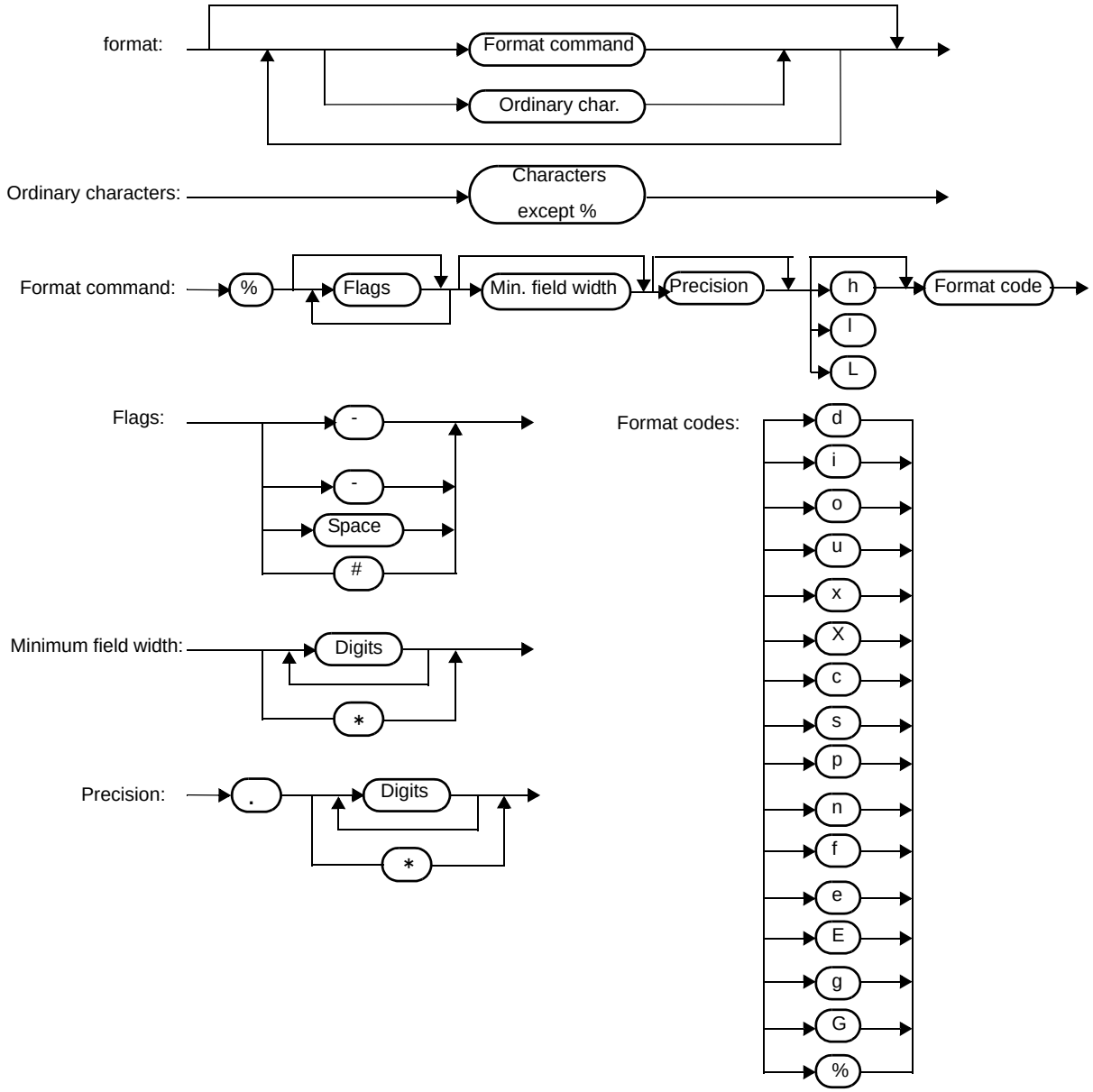
- Operations for invalid conversion specifiers are not guaranteed.
- When the actual argument is a union or a structure, or the pointer to indicate them (except the character type array in % s conversion or the pointer in % p conversion), operations are not guaranteed.
- The conversion result will not be truncated even when there is no field width or the field width is small. In other words, when the number of characters of the conversion result are larger than the field width, the field is extended to the width that includes the conversion result.
- The formats of the special output character string in %f, %e, %E, %g, %G conversions are shown below.

  non-numeric -> "(NaN)"
  +∞ -> "(+INF)"
  -∞ -> "(-INF)"

  sprintf writes a null character at the end of the string *s*. (This character is included in the return value count.)

The syntax of *format* commands is illustrated below.

Figure 6-2. Syntax of *format* Commands



- With the 78K0R C compiler,the near data pointers (2-byte length) must be specified when the medium model is used, and the far data pointers (4-byte length) must be specified when the large model is used for conversion specifiers s, p, and n that specify pointers as arguments.
  The function pointer length is fixed to 4 bytes in both models, but the pointer length in each model must be specified as a 2- or 4-byte length when using the pointer as an argument.

sscanf

Reads data from the input string according to a format

[Syntax]

```
#include <stdio.h>
int     sscanf ( const char *s , const char *format , ... ) ;
```

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to the input string *format* : Pointer to the string which indicates the input format commands ... : Pointer to object in which converted values are to be stored, and zero or more arguments	If the string *s* is empty: -1 If the string *s* is not empty : Number of assigned input data items

[Description]

- sscanf inputs data from the string pointed to by *s*. The string pointed to by *format* specifies the input string allowed for input.
  Zero or more arguments after *format* are used as pointers to an object. *format* specifies how data is to be converted from the input string.
- If there are insufficient arguments to match the format commands pointed to by *format*, proper operation by the compiler is not guaranteed. For excessive arguments, expression evaluation will be performed but no data will be input.
- The control string pointed to by *format* consists of zero or more format commands which are classified into the following 3 types:
  1 : Whitespace characters (one or more characters for which isspace becomes true)
  2 : Non-whitespace characters (other than %)
  3 : Format specifiers
- Each format specifier begins with the % character and is followed by these:

(1) Optional * character which suppresses assignment of data to the corresponding argument

(2) Optional decimal integer which specifies a maximum field width

(3) Optional h, l or L modifier which indicates the object size on the receiving side
If h precedes the d, i, o, or x format specifier, the argument is a pointer to not int but short int. If l precedes any of these format specifiers, the argument is a pointer to long int.
Likewise, if h precedes the u format specifier, the argument is a pointer to unsigned short int. If l precedes the u format specifier, the argument is a pointer to unsigned long int.
If l precedes the conversion specifier e, E, f, g, G, the argument is a pointer to double (a pointer to float in default without l). If L precedes, it is ignored.

Remark Conversion specifier: character to indicate the type of corresponding conversion (to be mentioned later)

- sscanf executes the format commands in "*format*" in sequence and if any format command fails, the function will terminate.

(1) A white-space character in the control string causes sscanf to read any number (including zero) of whitespace character up to the first non-white-space character (which will not be read). This white-space character command fails if it does not encounter any non-white-space character.

(2) A non-white-space character causes sscanf to read and discard a matching character. This command fails if the specified character is not found.

(3) The format commands define a collection of input streams for each type specifier (to be detailed later). The format commands are executed according to the following steps:

(a) The input white-space characters (specified by isspace) are skipped over, except when the type specifier is [, c, or n.

(b) The input data items are read from the string "*s*", except when the type specifier is n.
The input data items are defined as the longest input stream of the first partial stream of the string indicated by the type specifier (but up to the maximum field width if so specified). The character next to the input data items is interpreted as not have been read.
If the length of the input data items is 0, the format command execution fails.

(c) The input data items (number of input characters with the type specifier n) are converted to the type specified by the type specifier except the type specifier %.
If the input data items do not match with the specified type, the command execution fails.
Unless assignment is suppressed by *, the result of the conversion is stored in the object pointed to by the first argument which follows "*format*" and has not yet received the result of the conversion.

The following type specifiers are available:

Conversion Specifier	Contents
d	Converts a decimal integer (which may be signed). The corresponding argument must be a pointer to an integer.
I	Converts an integer (which may be signed). If a number is preceded by 0x or 0X, the number is interpreted as a hexadecimal integer. If a number is preceded by 0, the number is interpreted as an octal integer. Other numbers are regarded as decimal integers. The corresponding argument must be a pointer to an integer.
o	Converts an octal integer (which may be signed). The corresponding argument must be a pointer to an integer.
u	Converts an unsigned decimal integer. The corresponding argument must be a pointer to an unsigned integer.
x	Converts a hexadecimal integer (which may be signed).
e, E, f, g, G	Floating point value consists of optional sign (+ or -), one or more consecutive decimal number(s) including decimal point, optional exponent (e or E), and the following optional signed integer value. When overflow occurs as a result of conversion, or when underflow occurs with the conversion result $\pm\infty$, non-normalized number or $\pm$ 0 becomes the conversion result. The corresponding argument is a pointer to float.

Conversion Specifier	Contents
s	Input a character string consisting of a non-blank character string. The corresponding argument is a pointer to an integer. 0x or 0X can be allocated at the first hexadecimal integer. The corresponding argument must be a pointer an array that has sufficient size to accommodate this character string and a null terminator. The null terminator will be automatically added.
[	Inputs a character string consisting of expected character groups (called a scanset). The corresponding argument must be a pointer to the first character of an array that has sufficient size to accommodate this character string and a null terminator. The null terminator will be automatically added. The format commands continue from this character up to the closing square bracket ( )). The character string (called a scanlist) enclosed in the square brackets constitutes a scanset except when the character immediately after the opening square bracket is a circumflex ( ). When the character is a circumflex, all the characters other than a scanlist between the circumflex and the closing square bracket constitute a scanset. However, when a scanlist begins with [ ] or [^], this closing square bracket is contained in the scanlist and the next closing square list becomes the end of the scanlist. A hyphen (-) at other than the left or right end of a scanlist is interpreted as the punctuation mark for hyphenation if the character at the left of the range specifying hyphen (-) is not smaller than the right-hand character in ASCII code value.
c	Inputs a character string consisting of the number of characters specified by the field width. (If the field width specification is omitted, 1 is assumed.) The corresponding argument must be a pointer to the first character of an array that has sufficient size to accommodate this character string. The null terminator will not be added.
p	Reads an unsigned hexadecimal integer. The corresponding argument must be a pointer to void pointer.
n	Receives no input from the string *s*. The corresponding argument must be a pointer to an integer. The number of characters that are read thus far by this function from the string "s" is stored in the object that is pointed to by this pointer. The %n format command is not included in the return value assignment count.
%	Reads a % sign. Neither conversion nor assignment takes place.

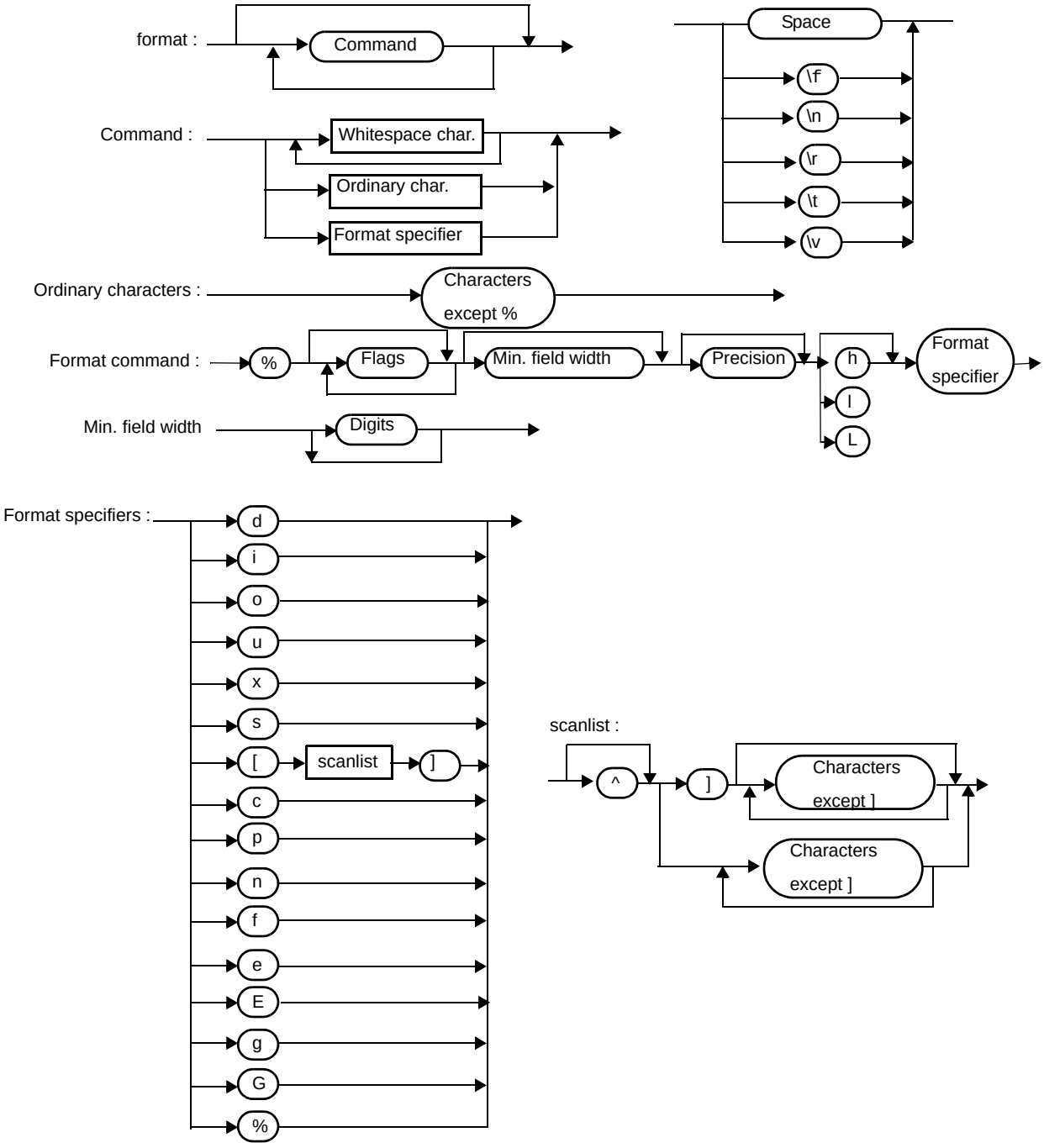
If a format specification is invalid, the format command execution fails.

If a null terminator appears in the input stream, sscanf will terminate.

If an overflow occurs in an integer conversion (with the d, i, o, u, x, or p format specifier), high-order bits will be truncated depending on the number of bits of the data type after the conversion.

The syntax of *format* commands is illustrated below.

Figure 6-3. Syntax of *format* Commands



- With the 78K0R C compiler, the near data pointers (2-byte length) must be specified when the medium model is used, and the far data pointers (4-byte length) must be specified when the large model is used for conversion specifiers s, p, and n that specify pointers as arguments.
  The function pointer length is fixed to 4 bytes in both models, but the pointer length in each model must be specified as a 2- or 4-byte length when using the pointer as an argument.

printf

Outputs data to SFR according to a format

[Syntax]

#include <stdio.h>
int printf (const char **format* , ...) ;

[Argument(s)/Return value]

Argument	Return Value
format : Pointer to the character string that indicates the output ... : 0 or more arguments to be converted	Number of character output to *s* (the null character at the end is not counted)

[Description]

- (0 or more) arguments following the *format* are converted and output using the putchar function, according to the output conversion specification specified in the *format*.
- The output conversion specification is 0 or more directives. Normal characters (other than the conversion specification starting with %) are output as is using the putchar function. The conversion specification is output using the putchar function by fetching and converting the following (0 or more) arguments.
- Each conversion specification is the same as that of the sprintf function.

scanf

Reads data from SFR according to a format

[Syntax]

```
#include <stdio.h>
int     scanf ( const char *format , ... ) ;
```

[Argument(s)/Return value]

Argument	Return Value
format : Pointer to the character string to indicate input conversion specification ... : Pointer (0 or more) argument to the object to assign the converted value	When the character string *s* is not null : Number of input items assigned

[Description]

- Performs input using getchar function. Specifies input string permitted by the character string *format* indicates. Uses the argument after the *format* as the pointer to an object. *format* specifies how the conversion is performed by the input string.
- When there are not enough arguments for the *format*, normal operation is not guaranteed. When the argument is excessive, the expression will be evaluated but not input.
- *format* consists of 0 or more directives. The directives are as follows.
  1 : One or more null character (character that makes isspace true)
  2 : Normal character (other than %)
  3 : Conversion indication
- If a conversion ends with a input character which conflicts with the input character, the conflicting input character is rounded down. The conversion indication is the same as that of the sscanf function.

vprintf

Outputs data to SFR according to a forma

[Syntax]

#include <stdio.h>
int vprintf (const char **format* , va_list *p*) ;

[Argument(s)/Return value]

Argument	Return Value
format: Pointer to the character string that indicates output conversion specification *p* : Pointer to the argument list	Number of output characters (the null character at the end is not counted)

[Description]

- The argument that the pointer of the argument list indicates is converted and output using putchar function according to the output conversion specification specified by the *format*.
- Each conversion specification is the same as that of sprintf function.

vsprintf

Writes data to a string according to a format

[Syntax]

#include <stdio.h>
int vsprintf (char *s , const char *format , va_list p) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to the character string that writes the output *format* : Pointer to the character string that indicates output conversion specification *p* : Pointer to the argument list	Number of characters output to *s* (the null character at the end is not counted)

[Description]

- Writes out the argument that the pointer of argument list indicates to the character strings which *s* indicates according to the output conversion specification specified by *format*.
- The output specification is the same as that of sprintf function.

getchar

Reads one character from SFR

[Syntax]

```
#include <stdio.h>
int         getchar ( void ) ;
```

[Argument(s)/Return value]

Argument	Return Value
None	A character read from SFR

[Description]

- Returns the value read from SFR symbol P0 (port 0).
- Error check related to reading is not performed.
- To change SFR to read, it is necessary either that the source be changed to be re-registered to the library or that the user create a new getchar function.

gets

Reads a string

[Syntax]

```
#include <stdio.h>
char  *gets ( char *s ) ;
```

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to input character string	Normal : s If the end of the file is detected without reading a character : Null pointer

[Description]

- Reads a character string using the getchar function and stores in the array that *s* indicates.
- When the end of the file is detected (getchar function returns -1) or when a line feed character is read, the reading of a character string ends. The line feed character read is abandoned, and a null character is written at the end of the character stored in the array in the end.
- When the return value is normal, it returns *s*.
- When the end of the file is detected and no character is read in the array, the contents of the array remains unchanged, and a null pointer is returned.

putchar

Outputs one character to SFR

[Syntax]

```
#include <stdio.h>
int         putchar ( int  c ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be output	Character to have been output

[Description]

- Writes the character specified by *c* to the SFR symbol P0 (port 0) (converted to unsigned char type).
- Error check related to writing is not performed.
- To change SFR to write, it is necessary either that the source is changed and re-registered to the library or the user create a new putchar function.

puts

Outputs a string

[Syntax]

```
#include <stdio.h>
int         puts ( const char  *s ) ;
```

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to an output character string	Normal : 0 When putchar function returns -1 : -1

[Description]

- Writes the character string indicated by *s* using putchar function, a line feed character is added at the end of the output.
- Writing of the null character at the end of the character string is not performed.
- When the return value is normal, 0 is returned, and when putchar function returns -1, -1 is returned.

__putc

Outputs one character to opaque

[Syntax]

```
#include <stdio.h>
int     __putc ( int c , void *opaque ) ;
```

[Argument(s)/Return value]

Argument	Return Value
c : Character to be output *opaque :* Pointer to a character output destination	Character to have been output

[Description]

- The __putc function writes the character specified by *c* (by converting it into the unsigned char type) to the destination indicated by *opaque*. The destination indicated by *opaque* is incremented by 1 byte.
- If *opaque* is 0, the putchar function is called and the return value of the putchar function is returned.

6.10 Utility Functions

The following utility functions are available.

Function Name	Purpose
atoi	Converts a decimal integer string to int
atol	Converts a decimal integer string to long
strtol	Converts a string to long
strtoul	Converts a string to unsigned long
calloc	Allocates an array's region and initializes it to zero
free	Releases a block of allocated memory
malloc	Allocates a block
realloc	Re-allocates a block
abort	Abnormally terminates the program
atexit	Registers a function to be called at normal termination
exit	Terminates the program
abs	Obtains the absolute value of an int type value
labs	Obtains the absolute value of a long type value
div	Performs int type division, obtains the quotient and remainder
ldiv	Performs long type division, obtains the quotient and remainder
brk	Sets the break value
sbrk	Increases/decreases the break value
atof	Converts a decimal integer string to double
strtod	Converts a string to double
itoa	Converts int to a string
ltoa	Converts long to a string
ultoa	Converts unsigned long to a string
rand	Generates a pseudo-random number
srand	Initializes the pseudo-random number generator state
bsearch	Binary search
qsort	Quick sort
strbrk	Sets the break value
strsbrk	Increases/decreases the break value
stritoa	Converts int to a string
strltoa	Converts long to a string
strultoa	Converts unsigned long to a string

atoi

Converts a decimal integer string to int

[Syntax]

#include <stdlib.h>
int atoi (const char **nptr*) ;

[Argument(s)/Return value]

Argument	Return Value
nptr : String to be converted	If converted properly : int value If positive overflow occurs : INT_MAX (32,767) If negative overflow occurs : INT_MIN (-32,768) IIf the string is invalid : 0

[Description]

- The atoi function converts the first part of the string pointed to by pointer *nptr* to an int value.
  The atoi function skips over zero or more white-space characters (for which isspace becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to an integer (until other than digits or a null character appears in the string). If no digits to convert is found in the string, the function returns 0.
  If an overflow occurs, the function returns INT_MAX (32,767) for positive overflow and INT_MIN (-32,768) for negative overflow.

atol

Converts a decimal integer string to long

[Syntax]

#include <stdlib.h>
long int atol (const char *_nptr_) ;

[Argument(s)/Return value]

Argument	Return Value
nptr : String to be converted	If converted properly : long int value If positive overflow occurs : LONG_MAX (2,147,483,647) If negative overflow occurs : LONG_MIN (-2,147,483,648) IIf the string is invalid : 0

[Description]

- The atol function converts the first part of the string pointed to by pointer *nptr* to a long int value.
  The atol function skips over zero or more white-space characters (for which isspace becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to an integer (until other than digits or null character appears in the string). If no digits to convert is found in the string, the function returns 0.
  If an overflow occurs, the function returns LONG_MAX (2,147,483,647) for positive overflow and LONG_MIN (-2,147,483,648) for negative overflow.

strtol

Converts a string to long

[Syntax]

#include <stdlib.h>
long int strtol (const char **nptr* , char ***endptr* , int *base*) ;

[Argument(s)/Return value]

Argument	Return Value
nptr : String to be converted *endptr* : Pointer storing pointer pointing to unrecognizable block *base* : Base for number represented in the string	If converted properly : long int value If positive overflow occurs : LONG_MAX (2,147,483,647) If negative overflow occurs : LONG_MIN (-2,147,483,648) If not converted : 0

[Description]

- The strtol function decomposes the string pointed by pointer *nptr* into the following 3 parts:

(1) String of whitespace characters that may be empty (to be specified by isspace)

(2) Integer representation by the *base* determined by the value of *base*

(3) String of one or more characters that cannot be recognized (including null terminators)

Remark The strtol function converts the part (2) of the string into an integer and returns this integer value.

- A *base* of 0 indicates that the *base* should be determined from the leading digits of the string. A leading 0x or 0X indicates a hexadecimal number; a leading 0 indicates an octal number; otherwise, the number is interpreted as decimal. (In this case, the number may be signed).
- If the *base* is 2 to 36, the set of letters from a to z or A to Z which can be part of a number (and which may be signed) with any of these bases are taken to represent 10 to 35.
  A leading 0x or 0X is ignored if the *base* is 16.
- If *endptr* is not a null pointer, a pointer to the part (3) of the string is stored in the object pointed to by *endptr*.
- If the correct value causes an overflow, the function returns LONG_MAX (2,147,483,647) for the positive overflow or LONG_MIN (-2,147,483,648) for the negative overflow depending on the sign and sets errno to ERANGE (2).
- If the string (2) is empty or the first non-white-space character of the string (ii) is not appropriate for an integer with the given *base*, the function performs no conversion and returns 0. In this case, the value of the string *nptr* is stored in the object pointed to by *endptr* (if it is not a null string). This holds true with the bases 0 and 2 to 36.

strtoul

Converts a string to unsigned long

[Syntax]

#include <stdlib.h>
unsigned long int strtoul (const char **nptr* , char ***endptr* , int *base*) ;

[Argument(s)/Return value]

Argument	Return Value
nptr : String to be converted *endptr* : Pointer storing pointer pointing to unrecognizable block *base* : Base for number represented in the string	If converted properly : unsigned long If overflow occurs : ULONG_MAX (4,294,967,295U) If not converted : 0

[Description]

- The strtoul function decomposes the string pointed by pointer *nptr* into the following 3 parts:

(1) String of white-space characters that may be empty (to be specified by isspace)

(2) Integer representation by the *base* determined by the value of *base*

(3) String of one or more characters that cannot be recognized (including null terminators)

Remark The strtoul function converts the part (2) of the string into a unsigned integer and returns this unsigned integer value.

- A *base* of 0 indicates that the *base* should be determined from the leading digits of the string. A leading 0x or 0X indicates a hexadecimal number; a leading 0 indicates an octal number; otherwise, the number is interpreted as decimal.
- If the *base* is 2 to 36, the set of letters from a to z or A to Z which can be part of a number (and which may be signed) with any of these bases are taken to represent 10 to 35. A leading 0x or 0X is ignored if the *base* is 16.
- If *endptr* is not a null pointer, a pointer to the part (3) of the string is stored in the object pointed to by *endptr*.
- If the correct value causes an overflow, the function returns ULONG_MAX (4,294,967,295U) and sets errno to ERANGE (2).
- If the string (2) is empty or the first non-white-space character of the string (2) is not appropriate for an integer with the given *base*, the function performs no conversion and returns 0. In this case, the value of the string *nptr* is stored in the object pointed to by *endptr* (if it is not a null string). This holds true with the bases 0 and 2 to 36.

calloc

Allocates an array's region and initializes it to zero

[Syntax]

#include <stdlib.h>
void * calloc (size_t *nmemb* , size_t *size*) ;

[Argument(s)/Return value]

Argument	Return Value
nmemb : Number of members in the array *size* : Size of each member	If the requested size is allocated : Pointer to the beginning of the allocated area If the requested size is not allocated : Null pointer

[Description]

- The calloc function allocates an area for an array consisting of n number of members (specified by *nmemb*), each of which has the number of bytes specified by *size* and initializes the area (array members) to zero.
- Returns the pointer to the beginning of the allocated area if the requested *size* is allocated.
- Returns the null pointer if the requested *size* is not allocated.
- The memory allocation will start from a break value and the address next to the allocated space will become a new break value. If the new break value is an odd number, the calloc function corrects it to be an even number. See " brk" for break value setting with the memory function brk.
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the calloc_n and calloc_f functions are not available.

free

Releases a block of allocated memory

[Syntax]

#include <stdlib.h>
void free (void *_ptr_) ;

[Argument(s)/Return value]

Argument	Return Value
ptr : Pointer to the beginning of block to be released	None

[Description]

- The free function releases the allocated space (before a break value) pointed to by *ptr*. (The malloc, calloc, or realloc called after the free will give you the space that was freed earlier.)
- If *ptr* does not point to the allocated space, the free will take no action. (Freeing the allocated space is performed by setting *ptr* as a new break value.)
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the free_n and free_f functions are not available.

malloc

Allocates a block

[Syntax]

#include <stdlib.h>
void *malloc (size_t *size*) ;

[Argument(s)/Return value]

Argument	Return Value
size : Size of memory block to be allocated	If the requested size is allocated : Pointer to the beginning of the allocated area If the requested size is not allocated : Null pointer

[Description]

- The malloc function allocates a block of memory for the number of bytes specified by *size* and returns a pointer to the first byte of the allocated area.
- If memory cannot be allocated, the function returns a null pointer.
- This memory allocation will start from a break value and the address next to the allocated area will become a new break value. If the new break value is an odd number, the malloc function corrects it to be an even number. See "brk" for break value setting with the memory function brk.
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the malloc_n and malloc_f functions are not available.

realloc

Re-allocates a block

[Syntax]

#include <stdlib.h>
void * realloc (void *_ptr_ , size_t _size_) ;

[Argument(s)/Return value]

Argument	Return Value
ptr : Pointer to the beginning of block previously allocated *size* : New size to be given to this block	If the requested size is reallocated : Pointer to the beginning of the reallocated space If *ptr* is a null pointer : Pointer to the beginning of the allocated space If the requested *size* is not reallocated or "*ptr*" is not a null pointer : Null pointer

[Description]

- The realloc function changes the size of the allocated space (before a break value) pointed to by *ptr* to that specified by *size*. If the value of *size* is greater than the size of the allocated space, the contents of the allocated space up to the original size will remain unchanged. The realloc function allocates only for the increased space. If the value of *size* is less than the size of the allocated space, the function will free the reduced space of the allocated space.
- If *ptr* is a null pointer, the realloc function will newly allocate a block of memory of the specified *size* (same as malloc).
- If *ptr* does not point to the block of memory previously allocated or if no memory can be allocated, the function executes nothing and returns a null pointer.
- Reallocation will be performed by setting the address of *ptr* plus the number of bytes specified by *size* as a new break value. If the new break value is an odd number, the realloc function corrects it to be an even number.
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the realloc_n and realloc_f functions are not available.

abort

Abnormally terminates the program

[Syntax]

```
#include <stdlib.h>
void abort ( void ) ;
```

[Argument(s)/Return value]

Argument	Return Value
None	No return to its caller.

[Description]

- The abort function loops and can never return to its caller.
- The user must create the abort processing routine.

atexit

Registers a function to be called at normal termination

[Syntax]

#include <stdlib.h>
int atexit (void (**func*) (void)) ;

[Argument(s)/Return value]

Argument	Return Value
func : Pointer to function to be registered	If function is registered as wrap-up function : 0 If function cannot be registered : 1

[Description]

- The atexit function registers the wrap-up function pointed to by *func* so that it is called without argument upon normal program termination by calling exit or returning from main.
- Up to 32 wrap-up functions may be established. If the warp-up function can be registered, atexit returns 0. If no more wrap-up function can be registered because 32 wrap-up functions have already been registered, the function returns 1.

exit

Terminates the program

[Syntax]

#include <stdlib.h>
void exit (int *status*) ;

[Argument(s)/Return value]

Argument	Return Value
status : Status value indicating termination	exit can never return.

[Description]

- The exit function causes immediate, normal termination of a program.
- This function calls the wrap-up functions in the reverse of the order in which they were registered with atexit.
- The exit function loops and can never return to its caller.
- The user must create the exit processing routine.

abs

Obtains the absolute value of an int type value

[Syntax]

#include <stdlib.h>
int abs (int *j*) ;

[Argument(s)/Return value]

Argument	Return Value
j : Any signed integer for which absolute value is to be obtained	If *j* falls within -32,767 <= *j* <= 32,767 : Absolute value of *j* If *j* is -32,768 : -32,768 (0x8000)

[Description]

- The abs returns the absolute value of its int type argument.
- If *j* is -32,768, the function returns -32,768.

labs

Obtains the absolute value of a long type value

[Syntax]

#include <stdlib.h>
long int labs (long int *j*) ;

[Argument(s)/Return value]

Argument	Return Value
j : Any signed integer for which absolute value is to be obtained	If *j* falls within -2,147,483,647 <= *j* <= 2,147,483,647 : Absolute value of *j* If the value of *j* is -2,147,483,648 : -2147483,648 (0x80000000)

[Description]

- The labs returns the absolute value of its long type argument.
- If the value of *j* is -2,147,483,648, the function returns -2,147,483,648.

div

Performs int type division, obtains the quotient and remainder

[Syntax]

#include <stdlib.h>
div_t div (int *numer* , int *denom*) ;

[Argument(s)/Return value]

Argument	Return Value
numer : Numerator of the division *denom* : Denominator of the division	Quotient to the quot element and the remainder to the rem element of div_t type member

[Description]

- The div function performs the integer division of numerator divided by denominator.
- The absolute value of quotient is defined as the largest integer not greater than the absolute value of *numer* divided by the absolute value of *denom*. The remainder always has the same sign as the result of the division (plus if *numer* and *denom* have the same sign; otherwise minus).
- The remainder is the value of *numer* - *denom* * quotient.
- If *denom* is 0, the quotient becomes 0 and the remainder becomes *numer*.
- If *numer* is -32,768 and *denom* is -1, the quotient becomes -32,768 and the remainder becomes 0.

ldiv

Performs long type division, obtains the quotient and remainder

[Syntax]

#include <stdlib.h>
ldiv_t ldiv (long int *numer* , long int *denom*) ;

[Argument(s)/Return value]

Argument	Return Value
numer : Numerator of the division *denom* : Denominator of the division	Quotient to the quot element and the remainder to the rem element of ldiv_t type member

[Description]

- The ldiv function performs the long integer division of numerator divided by denominator.
- The absolute value of quotient is defined as the largest long int type integer not greater than the absolute value of *numer* divided by the absolute value of *denom*. The remainder always has the same sign as the result of the division (plus if *numer* and *denom* have the same sign; otherwise minus).
- The remainder is the value of *numer* - *denom* * quotient.
- If *denom* is 0, the quotient becomes 0 and the remainder becomes *numer*.
- If *numer* is -2,147,483,648 and *denom* is -1, the quotient becomes -2,147,483,648 and the remainder becomes 0.

brk

Sets the break value

[Syntax]

```
#include <stdlib.h>
int brk ( char *endds ) ;
```

[Argument(s)/Return value]

Argument	Return Value
endds : Break value to be set block to be released	If converted properly : 0 If break value cannot be changed : -1

[Description]

- The brk function sets the value given by *endds* as a break value (the address next to the end address of an allocated block of memory).
- If *endds* is outside the permissible address range, the function sets no break value and sets errno to ENOMEM (3).
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the brk_n and brk_f functions are not available.

sbrk

Increases/decreases the break value

[Syntax]

#include <stdlib.h>
char *sbrk (int *incr*) ;

[Argument(s)/Return value]

Argument	Return Value
incr : Value (bytes) by which set break value is to be incremented/decremented.	If converted properly : Old break value If old break value cannot be incremented or decremented : -1

[Description]

- The sbrk function increments or decrements the set break value by the number of bytes specified by *incr*. (Increment or decrement is determined by the plus or minus sign of *incr*.)
- If an odd number is specified for *incr*, the sbrk function corrects it to be an even number.
- If the incremented or decremented break value is outside the permissible address range, the function does not change the original break value and sets errno to ENOMEM (3).
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the sbrk_n and sbrk_f functions are not available.

atof

Converts a decimal integer string to double

[Syntax]

#include <stdlib.h>
double atof (const char **nptr*) ;

[Argument(s)/Return value]

Argument	Return Value
nptr : String to be converted	If converted properly : Converted value If positive overflow occurs : HUGE_VAL (with sign of overflowed value) If negative overflow occurs : 0 IIf the string is invalid : 0

[Description]

- The atof function converts the string pointed to by pointer *nptr* to a double value.
  The atof function skips over zero or more whitespace characters (for which isspace becomes true) from the beginning of the string and converts the string from the character next to the skipped whitespaces to a floating point number (until other than digits or a null character appears in the string).
- A floating point number is returned when converted properly.
- If an overflow occurs on conversion, HUGE_VAL with the sign of the overflowed value is returned and ERANGE is set to errno.
- If valid digits are deleted due to an underflow or an overflow, a non-normalized number and +0 are returned respectively, and ERANGE is set to errno.
- If conversion cannot be performed, 0 is returned.

strtod

Converts a string to double

[Syntax]

#include <stdlib.h>
double strtod (const char **nptr* , char ***endptr*) ;

[Argument(s)/Return value]

Argument	Return Value
nptr : String to be converted *endptr* : Pointer storing pointer pointing to unrecognizable block	If converted properly : Converted value If positive overflow occurs : HUGE_VAL (with sign of overflowed value) If negative overflow occurs : 0 IIf the string is invalid : 0

[Description]

- The strtod function converts the string pointed to by pointer *nptr* to a double value.
  The strtod function skips over zero or more white-space characters (for which isspace becomes true) from the beginning of the string and converts the string from the character next to the skipped white-spaces to a floating-point number (until other than digits or null character appears in the string).
  If a character string starts with a character that does not satisfy this format, scanning is terminated. If *endptr* is not a null pointer, a pointer that starts with a character that may be a blank is stored in *endptr*.
- A floating-point number is returned when converted properly.
- If an overflow occurs on conversion, HUGE_VAL with the sign of the overflowed value is returned and ERANGE is set to errno.
- If valid digits are deleted due to an underflow or an overflow, a non-normalized number and +0 are returned respectively, and ERANGE is set to errno. In addition, *endptr* stores a pointer for next character string at that time.
- If conversion cannot be performed, 0 is returned.

itoa

Converts int to a string

[Syntax]

#include <stdlib.h>
char *itoa (int *value* , char **string* , int *radix*) ;

[Argument(s)/Return value]

Argument	Return Value
value : String to which integer is to be converted *string* : Pointer to the conversion result *radix* : Base of output string	If converted properly : Pointer to the converted string If not converted properly : Null pointer

[Description]

- The itoa, ltoa, and ultoa functions all convert the integer value specified by *value* to its string equivalent which is terminated with a null character and store the result in the area pointed to by "*string*".
- The *base* of the output string is determined by *radix*, which must be in the range 2 through 36. Each function performs conversion based on the specified *radix* and returns a pointer to the converted string. If the specified *radix* is outside the range 2 through 36, the function performs no conversion and returns a null pointer.

ltoa

Converts long to a string

[Syntax]

#include <stdlib.h>
char *ltoa (long *value* , char **string* , int *radix*) ;

[Argument(s)/Return value]

Argument	Return Value
value : String to which integer is to be converted *string* : Pointer to the conversion result *radix* : Base of output string	If converted properly : Pointer to the converted string If not converted properly : Null pointer

[Description]

- The itoa, ltoa, and ultoa functions all convert the integer value specified by *value* to its string equivalent which is terminated with a null character and store the result in the area pointed to by "*string*".
- The *base* of the output string is determined by *radix*, which must be in the range 2 through 36. Each function performs conversion based on the specified *radix* and returns a pointer to the converted string. If the specified *radix* is outside the range 2 through 36, the function performs no conversion and returns a null pointer.

ultoa

Converts unsigned long to a string

[Syntax]

#include <stdlib.h>
char *ultoa (unsigned long *value* , char **string* , int *radix*) ;

[Argument(s)/Return value]

Argument	Return Value
value : String to which integer is to be converted *string* : Pointer to the conversion result *radix* : Base of output string	If converted properly : Pointer to the converted string If not converted properly : Null pointer

[Description]

- The itoa, ltoa, and ultoa functions all convert the integer value specified by *value* to its string equivalent which is terminated with a null character and store the result in the area pointed to by "*string*".
- The *base* of the output string is determined by *radix*, which must be in the range 2 through 36. Each function performs conversion based on the specified *radix* and returns a pointer to the converted string. If the specified *radix* is outside the range 2 through 36, the function performs no conversion and returns a null pointer.

rand

Generates a pseudo-random number

[Syntax]

```
#include <stdlib.h>
int rand ( void ) ;
```

[Argument(s)/Return value]

Argument	Return Value
None	Pseudorandom integer in the range of 0 to RAND_MAX

[Description]

- Each time the rand function is called, it returns a pseudorandom integer in the range of 0 to RAND_MAX.

srand

Initializes the pseudo-random number generator state

[Syntax]

#include <stdlib.h>
void srand (unsigned int *seed*) ;

[Argument(s)/Return value]

Argument	Return Value
seed : Starting value for pseudorandom number generator	None

[Description]

- The srand function sets a starting value for a sequence of random numbers. *seed* is used to set a starting point for a progression of random numbers that is a return value when rand is called. If the same *seed* value is used, the sequence of pseudorandom numbers is the same when srand is called again.
- Calling rand before srand is used to set a *seed* is the same as calling rand after srand has been called with *seed* = 1. (The default *seed* is 1.)

bsearch

Binary search

[Syntax]

#include <stdlib.h>
void *bsearch (const void **key* , const void **base* , size_t *nmemb* , size_t *size* ,
int (**compare*) (const void *, const void *)) ;

[Argument(s)/Return value]

Argument	Return Value
key : Pointer to key for which search is made *base* : Pointer to sorted array which contains information to search *nmemb* : Number of array elements *size* : Size of an array *compare* : Pointer to function used to compare 2 keys	If the array contains the key : Pointer to the first member that matches "*key*" If the key is not contained in the array : Null pointer

[Description]

- The bsearch function performs a binary search on the sorted array pointed to by *base* and returns a pointer to the first member that matches the key pointed to by *key*. The array pointed to by *base* must be an array which consists of *nmemb* number of members each of which has the size specified by *size* and must have been sorted in ascending order.
- The function pointed to by compare takes 2 arguments (*key* as the 1st argument and array element as the 2nd argument), compares the 2 arguments, and returns:
  Negative value if the 1st argument is less than the 2nd argument
  0 if both arguments are equal
  Positive integer if the 1st argument is greater than the 2nd argument

qsort

Quick sort

[Syntax]

#include <stdlib.h>
void qsort (void *_base_ , size_t _nmemb_ , size_t _size_ , int (*_compare_) (const void * , const void *)) ;

[Argument(s)/Return value]

Argument	Return Value
base : Pointer to array to be sorted *nmemb* : Number of members in the array *size* : Size of an array member *compare* : Pointer to function used to compare 2 keys	None

[Description]

- The qsort function sorts the members of the array pointed to by *base* in ascending order.
  The array pointed to by *base* consists of *nmemb* number of members each of that has the size specified by *size*.
- The function pointed to by compare takes 2 arguments (array elements 1 and 2), compares the 2 arguments, and returns:
- The array element 1 as the 1st argument and array element 2 as the 2nd argument
  Negative value if the 1st argument is less than the 2nd argument
  0 if both arguments are equal
  Positive integer if the 1st argument is greater than the 2nd argument
- If the 2 array elements are equal, the element nearest to the top of the array will be sorted first.

strbrk

Sets the break value

[Syntax]

#include <stdlib.h>
int strbrk (char *_endds_) ;

[Argument(s)/Return value]

Argument	Return Value
endds : Break value to set	Normal : 0 When a break value cannot be changed : -1

[Description]

- Sets the value given by *endds* to the break value (the address following the address at the end of the area to be allocated).
- When *endds* is out of the permissible range, the break value is not changed. ENOMEM(3) is set to errno and -1 is returned.
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the strbrk_n and strbrk_f functions are not available.

strsbrk

Increases/decreases the break value

[Syntax]

#include <stdlib.h>
char *strsbrk (int *incr*) ;

[Argument(s)/Return value]

Argument	Return Value
incr : Amount to increase/decrease a break value	Normal : Old break value When a break value cannot be increased/decreased : -1

[Description]

- *incr* byte increases/decreases a break value (depending on the sign of *incr*).
- When the break value is out of the permissible range after increasing/decreasing, a break value is not changed. ENOMEM(3) is set to errno, and -1 is returned.
- Since the areas to be allocated by the 78K0R C compiler exist in the internal RAM, argument *ptr* is always the near pointer. Therefore, the strsbrk_n and strsbrk_f functions are not available.

stritoa

Converts int to a string

[Syntax]

#include <stdlib.h>
char *stritoa (int *value* , char **string* , int *radix*) ;

[Argument(s)/Return value]

Argument	Return Value
value : String to be converted *string* : Pointer to the conversion result *radix* : Radix to specify	Normal : Pointer to the converted character string Others : Null pointer

[Description]

- Converts the specified numeric value *value* to the character string that ends with a null character, and the result will be stored to the area specified with *string*. The conversion is performed by the *radix* specified, and the pointer to the converted character string will be returned.
- *radix* must be the value range between 2 to 36. In other cases, the conversion is not performed and a null pointer is returned.

strltoa

Converts long to a string

[Syntax]

#include <stdlib.h>
char *strltoa (long *value* , char **string* , int *radix*) ;

[Argument(s)/Return value]

Argument	Return Value
value : String to be converted *string* : Pointer to the conversion result *radix* : Radix to specify	Normal : Pointer to the converted character string Others : Null pointer

[Description]

- Converts the specified numeric value *value* to the character string that ends with a null character, and the result will be stored to the area specified with *string*. The conversion is performed by the *radix* specified, and the pointer to the converted character string will be returned.
- *radix* must be the value range between 2 to 36. In other cases, the conversion is not performed and a null pointer is returned.

strultoa

Converts unsigned long to a string

[Syntax]

#include <stdlib.h>
char *strultoa (unsigned long *value* , char **string* , int *radix*) ;

[Argument(s)/Return value]

Argument	Return Value
value : String to be converted *string* : Pointer to the conversion result *radix* : Radix to specify	Normal : Pointer to the converted character string Others : Null pointer

[Description]

- Converts the specified numeric value *value* to the character string that ends with a null character, and the result will be stored to the area specified with *string*. The conversion is performed by the *radix* specified, and the pointer to the converted character string will be returned.
- *radix* must be the value range between 2 to 36. In other cases, the conversion is not performed and a null pointer is returned.

6.11 String and Memory Functions

The following character string and memory functions are available.

Function Name	Purpose
memcpy	Copies a buffer for the specified number of characters
memmove	Copies a buffer for the specified number of characters
strcpy	Copies a string
strncpy	Copies the specified number of characters from the start of a string
strcat	Appends a string to a string
strncat	Appends the specified number of characters of a string to a string
memcmp	Compares the specified number of characters of two buffers
strcmp	Compares two strings
strncmp	Compares the specified number of characters of two strings
memchr	Searches for the specified string in the specified number of characters of a buffer
strchr	Searches for the specified character from within a string and returns the location of the first occurrence
strrchr	Searches for the specified character from within a string and returns the location of the last occurrence
strspn	Obtains the length from the start of a segment composed of only the characters included in the specified string within the string being searched
strcspn	Obtains the length from the start of a segment composed of characters other than those included in the specified string within the string being searched
strpbrk	Obtains the position of the first occurrence of any character in the specified string within the string being searched
strstr	Obtains the position of the first occurrence of the specified string within the string being searched
strtok	Decomposing character string into a string consisting of characters other than delimiters.
memset	Initializes the specified number of characters of a buffer with the specified character
strerror	Returns a pointer to the area that stores the error message string which corresponds to the specified error number
strlen	Obtains the length of a string
strcoll	Compares two strings based on region specific information
strxfrm	Transforms a string based on region specific information

memcpy

Copies a buffer for the specified number of characters

[Syntax]

#include <string.h>
void *memcpy (void **s1*, const void **s2* , size_t *n*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to object into which data is to be copied *s2* : Pointer to object containing data to be copied *n* : Number of characters to be copied	Value of *s1*

[Description]

- The memcpy function copies *n* number of consecutive bytes from the object pointed to by *s2* to the object pointed to by *s1*.
- If *s2* < *s1* < *s2* + *n* (*s1* and *s2* overlap), the memory copy operation by memcpy is not guaranteed (because copying starts in sequence from the beginning of the area).

memmove

Copies a buffer for the specified number of characters (Even if the buffer overlaps, the function performs memory copying properly)

[Syntax]

#include <string.h>
void *memmove (void **s1* , const void **s2* , size_t *n*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to object into which data is to be copied *s2* : Pointer to object containing data to be copied *n* : Number of characters to be copied	Value of *s1*

[Description]

- The memmove function also copies *n* number of consecutive bytes from the object pointed to by *s2* to the object pointed to by *s1*.
- Even if *s1* and *s2* overlap, the function performs memory copying properly.

strcpy

Copies a string

[Syntax]

#include <string.h>
char *strcpy (char **s1* , const char **s2*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to copy destination array *s2* : Pointer to copy source array	Value of *s1*

[Description]

- The strcpy function copies the contents of the character string pointed to by *s2* to the array pointed to by *s1* (including the terminating character).
- If *s2* < *s1* < (*s2* + Character length to be copied), the behavior of strcpy is not guaranteed (as copying starts in sequence from the beginning, not from the specified string).

strncpy

Copies the specified number of characters from the start of a string

[Syntax]

#include <string.h>
char *strncpy (char **s1* , const char **s2* , size_t *n*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to copy destination array *s2* : Pointer to copy source array *n* : Number of characters to be copied	Value of *s1*

[Description]

- The strncpy function copies up to the characters specified by *n* from the string pointed to by *s2* to the array pointed to by *s1*.
- If *s2* < *s1* < (*s2* + Character length to be copied or minimum value of *s2* + *n* - 1), the behavior of strncpy is not guaranteed (as copying starts in sequence from the beginning, not from the specified string).
- If the character string pointed to by *s2* is less than the number of characters specified by *n*, the strncpy function copies characters up to the terminating null character, and appends null characters until the number of copied characters reaches *n*.

strcat

Appends a string to a string

[Syntax]

#include <string.h>
char *strcat (char **s1* , const char **s2*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to a string to which a copy of another string (*s2*) is to be concatenated *s2* : Pointer to a string, copy of which is to be concatenated to another string (*s1*)	Value of *s1*

[Description]

- The strcat function concatenates a copy of the string pointed to by *s2* (including the null terminator) to the string pointed to by *s1*. The null terminator originally ending *s1* is overwritten by the first character of *s2*.
- When copying is performed between objects overlapping each other, the operation is not guaranteed.

strncat

Appends the specified number of characters of a string to a string

[Syntax]

#include <string.h>
char *strncat (char **s1* , const char **s2* , size_t *n*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to a string to which a copy of another string (*s2*) is to be concatenated *s2* : Pointer to a string, copy of which is to be concatenated to another string (*s1*) *n* : Number of characters to be concatenated	Value of *s1*

[Description]

- The strncat function concatenates not more than the characters specified by *n* of the string pointed to by *s2* (excluding the null terminator) to the string pointed to by *s1*. The null terminator originally ending *s1* is overwritten by the first character of *s2*.
- If the string pointed to by *s2* has fewer characters than specified by *n*, the strncat function concatenates the string including the null terminator. If there are more characters than specified by *n*, the *n* character section is concatenated starting from the top.
- The null terminator must always be concatenated.
- When copying is performed between objects overlapping each other, the operation is not guaranteed.

memcmp

Compares the specified number of characters of two buffers

[Syntax]

#include <string.h>
int memcmp (const void *s1 , const void *s2 , size_t n) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointers to 2 data objects to be compared *s2* : Pointers to 2 data objects to be compared *n* : Number of characters to compare	If the *n* characters of both *s1* and *s2* are compared and found to be the same : 0 If the *n* characters of both *s1* and *s2* are compared and found to be different : Value differences that converted the initial differing characters into int (*s1* letters - *s2* letters))

[Description]

- The memcmp function uses the *n* characters to compare the objects indicated by both *s1* and *s2*.
- The memcmp function returns 0, when the *n* characters of both *s1* and *s2* are compared and found to be the same.
- The memcmp function returns the value differences (*s1* letters - *s2* letters) that converted the initial differing characters into int if, the *n* characters of both *s1* and *s2* are compared and found to be different.

strcmp

Compares two strings

[Syntax]

#include <string.h>
int strcmp (const char *s1 , const char *s2) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to one string to be compared *s2* : Pointer to the other string to be compared	If *s1* is equal to *s2* : 0 If *s1* is less than or greater than *s2* : Value differences that converted the initial differing characters into int (*s1* letters - *s2* letters)

[Description]

- The strcmp function uses to compare the character strings indicated by both *s1* and *s2*.
- If *s1* is equal to *s2*, the function returns 0. If *s1* is less than or greater than *s2*, the strcmp function returns the value differences (*s1* letters - *s2* letters) that converted the initial diffrering characters into int.

strncmp

Compares the specified number of characters of two strings

[Syntax]

#include <string.h>
int strncmp (const char *s1 , const char *s2 , size_t n) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to one string to be compared *s2* : Pointer to the other string to be compared *n* : Number of characters to compare	If the *n* characters of both *s1* and *s2* are compared and found to be the same : 0 If the *n* characters of both *s1* and *s2* are compared and found to be different : Value differences that converted the initial differing characters into int (*s1* letters - *s2* letters)

[Description]

- The strncmp function uses the *n* characters to compare the objects indicated by both *s1* and *s2*.
- The strncmp function returns 0, when the *n* characters of both *s1* and *s2* are compared and found to be the same. The strncmp function returns the value differences (*s1* letters - *s2* letters) that converted the initial differing characters into int if, the *n* characters of both *s1* and *s2* are compared and found to be different.

memchr

Searches for the specified string in the specified number of characters of a buffer

[Syntax]

#include <string.h>
void *memchr (const void *s , int c , size_t n) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to objects in memory subject to search *c* : Character to be searched *n* : Number of bytes to be searched	If *c* is found : Pointer to the first occurrence of *c* If *c* is not found : Null pointer

[Description]

- The memchr function first converts the character specified by *c* to unsigned char and then returns a pointer to the first occurrence of this character within the *n* number of bytes from the beginning of the object pointed to by *s*.
- If the character is not found, the function returns a null pointer.

strchr

Searches for the specified character from within a string and returns the location of the first occurrence

[Syntax]

#include <string.h>
char *strchr (const char *s , int c) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to string to be searched *c* : Character specified for search	If *c* is found in *s* : Pointer indicating the first occurrence of *c* in string *s* If *c* is not found in *s* : Null pointer

[Description]

- The strchr function searches the string pointed to by *s* for the character specified by *c* and returns a pointer to the first occurrence of *c* (converted to char type) in the string.
- The null terminator is regarded as part of the string.
- If the specified character is not found in the string, the function returns a null pointer.

strrchr

Searches for the specified character from within a string and returns the location of the last occurrence

[Syntax]

#include <string.h>
char *strrchr (const char *s , int c) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to string to be searched *c* : Character specified for searchs	If *c* is found in *s* : Pointer indicating the last occurrence of *c* in string *s* If *c* is not found in *s* : Null pointer

[Description]

- The strrchr function searches the string pointed to by *s* for the character specified by *c* and returns a pointer to the last occurrence of *c* (converted to char type) in the string.
- The null terminator is regarded as part of the string.
- If no match is found, the function returns a null pointer.

strspn

Obtains the length from the start of a segment composed of only the characters included in the specified string within the string being searched

[Syntax]

#include <string.h>
size_t strspn (const char *s1 , const char *s2) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to string to be searched *s2* : Pointer to string whose characters are specified for match	Length of substring of the string *s1* that is made up of only those characters contained in the string *s2*

[Description]

- The strspn function returns the length of the substring of the string pointed to by *s1* that is made up of only those characters contained in the string pointed to by *s2*. In other words, this function returns the index of the first character in the string *s1* that does not match any of the characters in the string *s2*.
- The null terminator of *s2* is not regarded as part of *s2*.

strcspn

Obtains the length from the start of a segment composed of characters other than those included in the specified string within the string being searched

[Syntax]

```
#include <string.h>
size_t strcspn ( const char *s1 , const char *s2 ) ;
```

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to string to be searched *s2* : Pointer to string whose characters are specified for match	Length of substring of the string *s1* that is made up of only those characters not contained in the *s2*

[Description]

- The strcspn function returns the length of the substring of the string pointed to by *s1* that is made up of only those characters not contained in the string pointed to by *s2*. In other words, this function returns the index of the first character in the string *s1* that matches any of the characters in the string *s2*.
- The null terminator of *s2* is not regarded as part of *s2*.

strpbrk

Obtains the position of the first occurrence of any character in the specified string within the string being searched

[Syntax]

#include <string.h>
char *strpbrk (const char **s1* , const char **s2*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to string to be searched *s2* : Pointer to string whose characters are specified for match	If any match is found : Pointer to the first character in the string *s1* that matches any character in the string *s2* If no match is found : Null pointer

[Description]

- The strpbrk function returns a pointer to the first character in the string pointed to by *s1* that matches any character in the string pointed to by *s2*.
- If none of the characters in the string *s2* is found in the string *s1*, the function returns a null pointer.

strstr

Obtains the position of the first occurrence of the specified string within the string being searched

[Syntax]

#include <string.h>
char *strstr (const char **s1* , const char **s2*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to string to be searched *s2* : Pointer to specified string	If *s2* is found in *s1* : Pointer to the first appearance in the string *s1* of the string *s2* If *s2* is not found in *s1* : Null pointer If *s2* is a null string : Value of *s1*

[Description]

- The strstr function returns a pointer to the first appearance in the string pointed to by *s1* of the string pointed to by *s2* (except the null terminator of *s2*).
- If the string *s2* is not found in the string *s1*, the function returns a null pointer.
- If the string *s2* is a null string, the function returns the value of *s1*.

strtok

Obtains the length of a string

[Syntax]

#include <string.h>
char *strtok (char **s1* , const char **s2*) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to string from which tokens are to be obtained or null pointer *s2* : Pointer to string containing delimiters of token	If it is found : Pointer to the first character of a token If there is no token to return : Null pointer

[Description]

- A token is a string consisting of characters other than delimiters in the string to be specified.
- If *s1* is a null pointer, the string pointed to by the saved pointer in the previous strtok call will be decomposed. However, if the saved pointer is a null pointer, the function returns a null pointer without doing anything.
- If *s1* is not a null pointer, the string pointed to by *s1* will be decomposed.
- The strtok function searches the string pointed to by *s1* for any character not contained in the string pointed to by *s2*. If no character is found, the function changes the saved pointer to a null pointer and returns it. If any character is found, the character becomes the first character of a token.
- If the first character of a token is found, the function searches for any characters contained in the string *s2* after the first character of the token. If none of the characters is found, the function changes the saved pointer to a null pointer. If any of the characters is found, the character is overwritten by a null character and a pointer to the next character becomes a pointer to be saved.
- The function returns a pointer to the first character of the token.

memset

Initializes the specified number of characters of a buffer with the specified character

[Syntax]

#include <string.h>
void *memset (void *s , int c , size_t n) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to object in memory to be initialized *c* : Character whose value is to be assigned to each byte *n* : Number of bytes to be initialized	Value of *s*

[Description]

- The memset function first converts the character specified by *c* to unsigned char and then assigns the value of this character to the *n* number of bytes from the beginning of the object pointed to by *s*.

strerror

Returns a pointer to the area that stores the error message string which corresponds to the specified error number

[Syntax]

#include <string.h>
char *strerror (int *errnum*) ;

[Argument(s)/Return value]

Argument	Return Value
errnum : Error number	If message associated with error number exists : Pointer to string describing error message If no message associated with error number exists : Null pointer

[Description]

- The strerror function returns the following values associated with the value of *errnum*.

Value of *errnum*	Return Value
0	Pointer to the string "Error 0"
1 (EDOM)	Pointer to the string "Argument too large"
2 (ERANGE)	Pointer to the string "Result too large"
3 (ENOMEM)	Pointer to the string "Not enough memory"
Others	Null pointer

- Error message strings are allocated in a far area, so the return value is always a far pointer. This is why there are no strerror_n/strerror_f functions.

strlen

Obtains the length of a string

[Syntax]

#include <string.h>
size_t strlen (const char *s) ;

[Argument(s)/Return value]

Argument	Return Value
s : Pointer to character string	Length of string *s*

[Description]

- The strlen function returns the length of the null terminated string pointed to by *s*.

strcoll

Compares two strings based on region specific information

[Syntax]

```
#include <string.h>
int         strcoll ( const char *s1 , const char *s2 ) ;
```

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to comparison character string *s2* : Pointer to comparison character string	When character strings *s1* and *s2* are equal : 0 When character strings *s1* and *s2* are different : The difference between the values whose first different characters are converted to int (character of *s1* - character of *s2*)

[Description]

- The 78K0R C compiler does not support operations specific to cultural sphere.
  The operations are the same as that of strcmp.

strxfrm

Transforms a string based on region specific information

[Syntax]

#include <string.h>
size_t strxfrm (char *s1 , const char *s2 , size_t n) ;

[Argument(s)/Return value]

Argument	Return Value
s1 : Pointer to a compared character string *s2* : Pointer to a compared character string *n* : Maximum number of characters to *s1*	Returns the length of the character string of the result of the conversion (does not include a character string to indicate the end). If the returned value is *n* or more, the contents of the array indicated by *s1* is undefined.

[Description]

- The 78K0R C compiler does not support operations specific to cultural sphere.
  The operations are the same as those of the following functions.

```
strncpy ( s1 , s2 , c ) ;
return ( strlen ( s2 ) ) ;
```

6.12 Mathematical Functions

The following mathematical functions are available.

Function Name	Purpose
acos	Finds acos
asin	Finds asin
atan	Finds atan
atan2	Finds atan2
cos	Finds cos
sin	Finds sin
tan	Finds tan
cosh	Finds cosh
sinh	Finds sinh
tanh	Finds tanh
exp	Finds the exponential function
frexp	Finds mantissa and exponent part
ldexp	Finds x * 2 ^ exp
log	Finds the natural logarithm
log10	Finds the base 10 logarithm
modf	Finds the decimal and integer parts
pow	Finds yth power of x
sqrt	Finds the square root
ceil	Finds the smallest integer not smaller than x
fabs	Finds the absolute value of floating point number x
floor	Finds the largest integer not larger than x
fmod	Finds the remainder of x/y
matherr	Obtains the exception processing for the library handling floating point numbers
acosf	Finds acos
asinf	Finds asin
atanf	Finds atan
atan2f	Finds atan of y/x
cosf	Finds cos
sinf	Finds sin
tanf	Finds tan
coshf	Finds cosh
sinhf	Finds sinh
tanhf	Finds tanh
expf	Finds the exponential function
frexpf	Finds mantissa and exponent part
ldexpf	Finds x * 2 ^ exp
logf	Finds the natural logarithm

Function Name	Purpose
log10f	Finds the base 10 logarithm
modff	Finds the decimal and integer parts
powf	Finds yth power of x
sqrtf	Finds the square root
ceilf	Finds the smallest integer not smaller than x
fabsf	Finds the absolute value of floating point number x
floorf	Finds the largest integer not larger than x
fmodf	Finds the remainder of x/y

acos

Finds acos

[Syntax]

#include <math.h>
double acos (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	When -1 <= *x* <= 1 : acos of *x* When *x* < -1, 1 < *x*, *x* = NaN : NaN

[Description]

- Calculates acos of *x* (range between 0 and π).
- In the case of the definition area error of *x* < -1, 1 < *x*, NaN is returned and EDOM is set.
- When *x* is non-numeric, NaN is returned.

asin

Finds asin

[Syntax]

```
#include <math.h>
double  asin ( double x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	When -1 <= *x* <= 1 : asin of *x* When *x* < -1, 1 < *x*, *x* = NaN : NaN When *x* = -0 : -0 When underflow occurs : Non-normalized number

[Description]

- Calculates asin (range between $-\pi/2$ and $+\pi/2$) of *x*.
- In the case of area error of *x* < -1, 1 < *x*, NaN is returned and EDOM is set to errno.
- When *x* is non-numeric, NaN is returned.
- When *x* is -0, -0 is returned.
- If underflow occurs as a result of conversion, a non-normalized number is returned.

atan

Finds atan

[Syntax]

#include <math.h>
double atan (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : atan of *x* When *x* = NaN : NaN When *x* = -0 : -0 When underflow occurs : Non-normalized number

[Description]

- Calculates atan (range between $-\pi/2$ and $+\pi/2$) of *x*.
- When *x* is non-numeric, NaN is returned.
- When *x* is -0, -0 is returned.
- If underflow occurs as a result of conversion, a non-normalized number is returned.

atan2

Finds atan of y/x

[Syntax]

#include <math.h>
double atan2 (double *y* , double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *y* : Numeric value to perform operation	Normal : atan of *y*/*x* When both *x* and *y* are 0 or y/x is the value that cannot be expressed, or either *x* or *y* is NaN and both *x* and *y* are $\pm\infty$: NaN When underflow occurs : Non-normalized number

[Description]

- atan (range between $-\pi$ and $+\pi$) of y/x is calculated.
- When both *x* and *y* are 0 or y/x is the value that cannot be expressed, or when both *x* and *y* are infinite, NaN is returned and EDOM is set to errno.
- If either *x* or *y* is non-numeric, NaN is returned.
- If underflow occurs as a result of operation, non-normalized number is returned.

cos

Finds cos

[Syntax]

#include <math.h>
double cos (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : cos of *x* When *x* = NaN, when *x* is infinite : NaN

[Description]

- Calculates cos of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If the absolute value of *x* is extremely large, the result of an operation becomes an almost meaningless value.

sin

Finds sin

[Syntax]

#include <math.h>
double sin (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : sin of *x* When *x* = NaN, when *x* is infinite : NaN When underflow occurs : Non-normalized number

[Description]

- Calculates sin of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of *x* is extremely large, the result of an operation becomes an almost meaningless value.

tan

Finds tan

[Syntax]

#include <math.h>
double tan (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : tan of *x* When *x* = NaN, *x* = ± ∞ : NaN When underflow occurs : Non-normalized number

[Description]

- Calculates tan of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of *x* is extremely large, the result of an operation becomes an almost meaningless value.

cosh

Finds cosh

[Syntax]

```
#include <math.h>
double  cosh ( double x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : cosh of *x* When *x* = NaN : NaN When *x* = ± ∞: +∞ When overflow occurs HUGE_VAL (with the sign of the overflown value)

[Description]

- Calculates cosh of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, a positive infinite value is returned.
- If overflow occurs as a result of operation, HUGE_VAL with a positive sign is returned, and ERANGE is set to errno.

sinh

Finds sinh

[Syntax]

#include <math.h>
double sinh (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : sinh of *x* When *x* = NaN : NaN When *x* = ± ∞ : ± ∞ When overflow occurs : HUGE_VAL (with the sign of the overflown value) When underflow occurs : ± 0

[Description]

- Calculates sinh of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± ∞, ± ∞ is returned.
- If overflow occurs as a result of operation, HUGE_VAL with the sign of the overflown value is returned, and ERANGE is set to errno.
- If underflow occurs as a result of operation, +0 is returned.

tanh

Finds tanh

[Syntax]

```
#include <math.h>
double  tanh ( double x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : tanh of *x* When *x* = NaN : NaN When *x* = ± ∞ : ± 1 When underflow occurs : ± 0

[Description]

- Calculates tanh of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± ∞, ± 1 is returned.
- If underflow occurs as a result of operation, ± 0 is returned.

exp

Finds the exponential function

[Syntax]

#include <math.h>
double exp (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : Exponent function of *x* When *x* = NaN : NaN When *x* = +∞ : +∞ When *x* = -∞ : +0 When underflow occurs : Non-normalized number When annihilation of valid digits occurs due to underflow : +0 When overflow occurs : HUGE_VQAL (with positive sign)

[Description]s

- Calculates exponent function of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is +∞, +∞ is returned.
- If *x* is -∞, +0 is returned.
- If underflow occurs as a result of operation, non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of operation, +0 is returned.
- If overflow occurs as a result of operation, HUGE_VAL with a positive sign is returned and ERANGE is set to errno.

frexp

Finds mantissa and exponent part

[Syntax]

#include <math.h>
double frexp (double *x* , int **exp*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *exp* : Pointer to store exponent part	Normal : Mantissa of *x* When *x* = NaN, *x* = ± ∞: NaN When *x* = ± 0 : ± 0

[Description]

- Divide a floating point number *x* to mantissa m and exponent n such as *x* = m * 2 ^ n and returns mantissa m.
- Exponent n is stored where the pointer *exp* indicates. The absolute value of m, however, is 0.5 or more and less than 1.0.
- If *x* is non-numeric, NaN is returned and the value of **exp* is 0.
- If *x* is infinite, NaN is returned, and EDOM is set to errno with the value of **exp* as 0.
- If *x* is ± 0, ± 0 is returned and the value of **exp* is 0.

ldexp

Finds x * 2 ^ exp

[Syntax]

#include <math.h>
double ldexp (double *x* , int *exp*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *exp* : Exponentiation	Normal : *x* * 2 ^ *exp* When *x* = NaN : NaN When *x* = ± ∞ : ± ∞ When *x* = ± 0 : ± 0 When overflow occurs : HUGE_VAL (with the sign of the overflown value) When underflow occurs : Non-normalized number When annihilation of valid digits occurs due to underflow : ± 0

[Description]

- Calculates x * 2 ^ exp.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± ∞, ± ∞ is returned.
- If *x* is ± 0, ± 0 is returned.
- If overflow occurs as a result of operation, HUGE_VAL with the overflown value is returned and ERANGE is set to errno.
- If underflow occurs as a result of operation, non-normalized number is returned.
- If annihilation of valid digits due to underflow occurs as a result of operation, ± 0 is returned.

log

Finds the natural logarithm

[Syntax]

#include <math.h>
double log (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : Natural logarithm of *x* When *x* < 0 : NaN When *x* = 0 : -∞ When *x* = NaN : NaN When *x* is infinite : +∞

[Description]

- Finds natural logarithm of *x*.
- In the case of area error of *x* < 0, NaN is returned, EDOM is set to errno.
- If *x* = 0, -∞ is returned, and ERANGE is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* is +∞, +∞ is returned.

log10

Finds the base 10 logarithm

[Syntax]

#include <math.h>
double log10 (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : ithm with 10 of *x* as the base When *x* < 0 : NaN When *x* = 0 : -∞ When *x* = NaN : NaN When *x* is infinite : +∞

[Description]

- Finds logarithm with 10 of *x* as the base.
- In the case of area error of *x* < 0, NaN is returned, EDOM is set to errno.
- If *x* = 0, -∞ is returned, and ERANGE is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* is +∞, +∞ is returned.

modf

Finds the decimal and integer parts

[Syntax]

#include <math.h>
double modf (double *x* , double **iptr*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *iptr* : Pointer to integer part	Normal : Fraction part of *x* When *x* is non-numeric or infinite : NaN When *x* is ± 0 : ± 0

[Description]

- Divides a floating point number *x* to fraction part and integer part
- Returns fraction part with the same sign as that of *x*, and stores the integer part to the location indicated by the pointer *iptr*.
- If *x* is non-numeric, NaN is returned and stored to the location indicated by the pointer *iptr*.
- If *x* is infinite, NaN is returned and stored to the location indicated by the pointer *iptr*, and EDOM is set to errno.
- If *x* = ± 0, ± 0 is stored to the location indicated by the pointer *iptr*.

pow

Finds yth power of x

[Syntax]

#include <math.h>
double pow (double *x* , double *y*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *y* : Multiplier	Normal : *x* ^ *y* Either when *x* = NaN or *y* = NaN, *x* = +∞ and *y* = 0, *x* < 0 and *y* ≠ integer, *x* < 0 andy = ± ∞, *x* = 0 and *y* <= 0 : NaN When overflow occurs : HUGE_VAL (with the sign of overflown value) When underflow occurs : Non-normalized number When annihilation of valid digits occurs due to underflow : ± 0

[Description]

- Calculates x ^ y.
- When *x* = NaN or *y* = NaN, NaN is returned.
- Either when *x* = ± ∞ and *y* = 0, *x* < 0 and *y* ≠ integer, *x* < 0 and *y* = ± ∞ or *x* = 0 and *y* <= 0, NaN is returned and EDOM is set to errno.
- If overflow occurs as a result of operation, HUGE_VAL with the sign of overflown value is returned, and ERANGE is set to errno.
- If underflow occurs, a non-normalized number is returned.
- If annihilation of valid digits occurs due to underflow, ± 0 is returned.

sqrt

Finds the square root

[Syntax]

```
#include <math.h>
double  sqrt ( double x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	When *x*>=0 : Square root of *x* When *x* < 0 : 0 When *x* = NaN : NaN When *x* = ± 0 : ± 0

[Description]

- Calculates the square root of *x*.
- In the case of area error of *x* < 0, 0 is returned and EDOM is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± 0, ± 0 is returned.

ceil

Finds the smallest integer not smaller than x

[Syntax]

#include <math.h>
double ceil (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : The minimum integer no less than *x* When *x* is non-numeric or when *x* is infinite : NaN When *x* = -0 : +0 When the minimum integer no less than *x* cannot be expressed : x

[Description]

- Finds the minimum integer no less than *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If *x* is -0, +0 is returned.
- If the minimum integer no less than *x* cannot be expressed, *x* is returned.

fabs

Finds the absolute value of floating point number x

[Syntax]

```
#include <math.h>
double  fabs ( double  x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to find the absolute value	Normal : Absolute value of *x* When *x* = NaN : NaN When *x* = -0 : +0

[Description]

- Finds the absolute value of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is -0, +0 is returned.

floor

Finds the largest integer not larger than x

[Syntax]

#include <math.h>
double floor (double *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : The maximum integer no more than *x* When *x* is non-numeric or when *x* is infinite : NaN When *x* = -0 : +0 When the maximum integer no more than *x* cannot be expressed : x

[Description]

- Finds the maximum integer no more than *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If *x* is -0, +0 is returned.
- If the maximum integer no more than *x* cannot be expressed, *x* is returned.

fmod

Finds the remainder of x/y

[Syntax]

#include <math.h>
double fmod (double *x* , double *y*)

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *y* : Numeric value to perform operation	Normal : Remainder of x/y When *x* is non-numeric or *y* is non-numeric, when *y* is $\pm$ 0, when *x* is $\pm \infty$: NaN When $x \neq \infty$ and $y = \pm \infty$: x

[Description]

- Calculates the remainder of x/y expressed with x - i * y. i is an integer.
- If $y \neq 0$, the return value has the same sign as that of *x* and the absolute value is less than that of *y*.
- If *x* is non-numeric or *y* is non-numeric, NaN is returned.
- If *y* is $\pm$ 0 or $x = \pm \infty$, NaN is returned and EDOM is set to errno.
- If *y* is infinite, *x* is returned unless *x* is infinite.

matherr

Obtains the exception processing for the library handling floating point numbers

[Syntax]

#include <math.h>
void matherr (struct exception *x) ;

[Argument(s)/Return value]

Argument	Return Value

`struct exception {`
`int type ;`
`char *name ;`
`}`
 	
type :
Numeric value to indicate arithmetic exception
name :
Function name
	None

[Description]

- When an exception is generated, matherr is automatically called in the standard library and run-time library that deal with floating-point numbers.
- When called from the standard library, EDOM and ERANGE are set to errno.
  The following shows the relationship between the arithmetic exception type and errno.

Type	Arithmetic Exception	Value Set to errno
1	Underflow	ERANGE
2	Annihilation	ERANGE
3	Overflow	ERANGE
4	Zero division	EDOM
5	Inoperable	EDOM

  Original error processing can be performed by changing or creating matherr.
- The argument is always a near pointer, because it points to an exception structure in internal RAM. This is why there are no matherr_n/matherr_f functions.

acosf

Finds acos

[Syntax]

```
#include <math.h>
float  acosf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	When -1 <= *x* <= 1 : acos of *x* When *x* < -1, 1 < *x*, *x* = NaN : NaN

[Description]

- Calculates acos (range between 0 and π) of *x*.
- In the case of definition area error of *x* < -1, 1 < *x*, NaN is returned and EDOM is set to errno.
- If *x* is non-numeric, NaN is returned.

asinf

Finds asin

[Syntax]

```
#include <math.h>
float  asinf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	When -1 <= *x* <= 1 : asin of *x* When *x* < -1, 1 < *x*, *x* = NaN : NaN When *x* = -0 : -0 When underflow occurs : Non-normalized number

[Description]

- Calculates asin (range between $-\pi/2$ and $+\pi/2$) of *x*.
- In the case of definition area error of *x* < -1, 1 < *x*, NaN is returned and EDOM is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* = -0, -0 is returned.
- If underflow occurs as a result of operation, a non-normalized number is returned.

atanf

Finds atan

[Syntax]

```
#include <math.h>
float  atanf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : atan of *x* When *x* = NaN : NaN When *x* = -0 : -0 When underflow occurs : Non-normalized number

[Description]

- Calculates atan (range between $-\pi/2$ and $+\pi/2$) of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* = -0, -0 is returned.
- If underflow occurs as a result of operation, a non-normalized number is returned.

atan2f

Finds atan of y/x

[Syntax]

```
#include <math.h>
float  atan2f ( float y , float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *y* : Numeric value to perform operation	Normal : atan of y/x When both *x* and *y* are 0 or a value whose y/ *x* cannot be expressed, or either *x* or *y* is NaN, both *x* and *y* are infinite : NaN When underflow occurs : Non-normalized number

[Description]

- Calculates atan (range between $-\pi$ and $+\pi$) of y/x.
- When both *x* and *y* are 0 or the value whose y/x cannot be expressed, or when both *x* and *y* are infinite, NaN is returned and EDOM is set to errno.
- When either *x* or *y* is non-numeric, NaN is returned.
- If underflow occurs as a result of operation, a non-normalized number is returned.

cosf

Finds cos

[Syntax]

```
#include <math.h>
float  cosf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : cos of *x* When *x* = NaN, *x* is infinite : NaN

[Description]

- Calculates cos of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If the absolute value of *x* is extremely large, the result of an operation becomes an almost meaningless value.

sinf

Finds sin

[Syntax]

```
#include <math.h>
float  sinf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : sin of *x* When *x* = NaN, *x* is infinite : NaN When underflow occurs : Non-normalized number

[Description]

- Calculates sin of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of *x* is extremely large, the result of an operation becomes an almost meaningless value.

tanf

Finds tan

[Syntax]

```
#include <math.h>
float  tanf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : tan of *x* When *x* = NaN, *x* is infinite : NaN When underflow occurs : Non-normalized number

[Description]

- Calculates tan of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If underflow occurs as a result of operation, a non-normalized number is returned.
- If the absolute value of *x* is extremely large, the result of an operation becomes an almost meaningless value.

coshf

Finds cosh

[Syntax]

#include <math.h>
float coshf (float *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : cosh of *x* When *x* = NaN : NaN When *x* is infinite : +∞ When overflow occurs : HUGE_VQAL (with positive sign)

[Description]

- Calculates cosh of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, positive infinite value is returned.
- If overflow occurs as a result of operation, HUGE_VAL with a positive sign is returned and ERANGE is set to errno.

sinhf

Finds sinh

[Syntax]

```
#include <math.h>
float  sinhf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : sinh of *x* When *x* = NaN : NaN When *x* = ± ∞: ± ∞ When overflow occurs : HUGE_VAL (with a sign of the overflown value) When underflow occurs : ± 0

[Description]

- Calculates sinh of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± ∞, ± ∞ is returned.
- If overflow occurs as a result of operation, HUGE_VAL with the sign of overflown value is returned and ERANGE is set to errno.
- If underflow occurs as a result of operation, +± 0 is returned.

tanhf

Finds tanh

[Syntax]

```
#include <math.h>
float  tanhf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : tanh of *x* When *x* = NaN : NaN When *x* = ± ∞ : ± 1 When underflow occurs : ± 0

[Description]

- Calculates tanh of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± ∞, ± 1 is returned.
- If underflow occurs as a result of operation, ± 0 is returned.

expf

Finds the exponential function

[Syntax]

```
#include <math.h>
float  expf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : Exponent function of *x* When *x* = NaN : NaN When *x* = +∞: +∞ When *x* = -∞ : +0 When overflow occurs : HUGE_VQAL (with positive sign) When underflow occurs : Non-normalized number When annihilation of valid digits occurs due to underflow : +0

[Description]

- Calculates exponent function of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is +∞, +∞ is returned.
- If *x* is -∞, +0 is returned.
- If overflow occurs as a result of operation, HUGE_VAL with a positive sign is returned and ERANGE is set to errno.
- If underflow occurs as a result of operation, non-normalized number is returned.
- If annihilation of effective digits occurs due to underflow as a result of operation, +0 is returned.

frexpf

Finds mantissa and exponent part

[Syntax]

#include <math.h>
float frexpf (float *x* , int **exp*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *exp* : Pointer to store exponent part	Normal : Mantissa of *x* When *x* = NaN, *x* = ± ∞ : NaN When *x* = ± 0 : ± 0

[Description]

- Divides a floating-point number *x* to mantissa m and exponent n such as *x* = m * 2 ^ n and returns mantissa m.
- Exponent n is stored in where the pointer *exp* indicates. The absolute value of m, however, is 0.5 or more and less than 1.0.
- If *x* is non-numeric, NaN is returned and the value of **exp* is 0.
- If *x* is ± ∞, NaN is returned, and EDOM is set to errno with the value of **exp* as 0.
- If *x* is ± 0, ± 0 is returned and the value of **exp* is 0.

ldexpf

Finds x * 2 ^ exp

[Syntax]

#include <math.h>
float ldexpf (float *x* , int *exp*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *exp* : Exponentiation	Normal : *x* * 2 ^ *exp* When *x* = NaN : NaN When *x* = ± ∞ : ± ∞ When *x* = ± 0 : ± 0 When overflow occurs : HUGE_VAL (with the sign of overflown value) When underflow occurs : Non-normalized number When annihilation of valid digits occurs due to underflow : ± 0

[Description]

- Calculates x * 2 ^ exp.
- If *x* is non-numeric, NaN is returned. If *x* is ± ∞, ± ∞ is returned. If *x* is ± 0, ± 0 is returned.
- If overflow occurs as a result of operation, HUGE_VAL with the sign of overflown value is returned and ERANGE is set to errno.
- If underflow occurs as a result of operation, non-normalized number is returned .
- If annihilation of valid digits due to underflow occurs as a result of operation, ± 0 is returned.

logf

Finds the natural logarithm

[Syntax]

#include <math.h>
float logf (float *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : Natural logarithm of *x* When *x* < 0 : NaN When *x* = 0 : -∞ When *x* = NaN : NaN When *x* is infinite : +∞

[Description]

- Finds natural logarithm of *x*.
- In the case of area error of *x* < 0, NaN is returned, and EDOM is set to errno.
- If *x* = 0, -∞ is returned, and ERANGE is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* is +∞, +∞ is returned.

log10f

Finds the base 10 logarithm

[Syntax]

```
#include <math.h>
float  log10f ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : Logarithm with 10 of *x* as the base When *x* < 0 : NaN When *x* = 0 : -∞ When *x* = NaN : NaN When *x* = +∞: +∞

[Description]

- Finds logarithm with 10 of *x* as the base.
- In the case of area error of *x* < 0, NaN is returned, and EDOM is set to errno.
- If *x* = 0, -∞ is returned, and ERANGE is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* is +∞, +∞ is returned.

modff

Finds the decimal and integer parts

[Syntax]

#include <math.h>
float modff (float *x* , float **iptr*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *iptr* : Pointer to integer part	Normal : Fraction part of *x* When *x* = NaN, *x* is infinite : NaN When *x* = ± 0 : ± 0

[Description]

- Divides a floating point number *x* to fraction part and integer part.
- Returns fraction part with the same sign as that of *x*, and stores integer part to location indicated by the pointer *iptr*.
- If *x* is non-numeric, NaN is returned and stored location indicated by the pointer *iptr*.
- If *x* is infinite, NaN is returned and stored location indicated by the pointer *iptr*, and EDOM is set to errno.
- If *x* = ± 0, ± 0 is returned and stored location indicated by the pointer *iptr*.

powf

Finds yth power of x

[Syntax]

```
#include <math.h>
float  powf ( float x , float y ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *y* : Multiplier	Normal : *x* ^ *y* Either when *x* = NaN or *y* = NaN *x* = +∞ and *y* = 0 *x* < 0 and *y* ≠ integer, *x* < 0 and *y* = ± ∞ *x* = 0 and *y* <= 0 : NaN When overflow occurs : HUGE_VAL (with the sign of overflown value) When underflow occurs : Non-normalized number When annihilation of valid digits occurs due to underflow : ± 0

[Description]

- Calculates x ^ y.
- When *x* = NaN or *y* = NaN, NaN is returned.
- Either when *x* = +∞ and *y* = 0, *x* < 0 and *y* ≠ integer, *x* < 0 and *y* = ± ∞, or *x* = 0 and *y* <= 0, NaN is returned and EDOM is set to errno.
- If overflow occurs as a result of operation, HUGE_VAL with the sign of overflown value is returned, and ERANGE is set to errno.
- If underflow occurs, a non-normalized number is returned.
- If annihilation of valid digits occurs due to underflow, ± 0 is returned.

sqrtf

Finds the square root

[Syntax]

```
#include <math.h>
float  sqrtf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	When *x*>=0 : Square root of *x* When *x* < 0 : 0 When *x* = NaN : NaN When *x* = ± 0 : ± 0

[Description]

- Calculates the square root of *x*.
- In the case of area error of *x* < 0, 0 is returned and EDOM is set to errno.
- If *x* is non-numeric, NaN is returned.
- If *x* is ± 0, ± 0 is returned.

ceilf

Finds the smallest integer not smaller than x

[Syntax]

```
#include <math.h>
float  ceilf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : The minimum integer no less than *x* When *x* = NaN, *x* is infinite : NaN When *x* = -0 : +0 When the minimum integer no less than *x* cannot be expressed : *x*

[Description]

- Finds the minimum integer no less than *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is -0, +0 is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If the minimum integer no less than *x* cannot be expressed, *x* is returned.

fabsf

Finds the absolute value of floating point number x

[Syntax]

#include <math.h>
float fabsf (float *x*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to find the absolute value	Normal : Absolute value of *x* When *x* is non-numeric : NaN When *x* = -0 : +0

[Description]

- Finds the absolute value of *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is -0, +0 is returned.

floorf

Finds the largest integer not larger than x

[Syntax]

```
#include <math.h>
float  floorf ( float x ) ;
```

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation	Normal : The maximum integer no more than *x* When *x* = NaN, *x* is infinite : NaN When *x* = -0 : +0 When the maximum integer no more than *x* cannot be expressed : x

[Description]

- Finds the maximum integer no more than *x*.
- If *x* is non-numeric, NaN is returned.
- If *x* is infinite, NaN is returned and EDOM is set to errno.
- If *x* is -0, +0 is returned.
- If the maximum integer no more than *x* cannot be expressed, *x* is returned.

fmodf

Finds the remainder of x/y

[Syntax]

#include <math.h>
float fmodf (float *x* , float *y*) ;

[Argument(s)/Return value]

Argument	Return Value
x : Numeric value to perform operation *y* : Numeric value to perform operation	Normal : Remainder of x/y When *y* is ± 0 or *x* is ± ∞, When *x* is non-numeric or *y* is non-numeric : NaN When $x \neq \infty$ and $y = \pm \infty$: x

[Description]

- Calculates the remainder of x/y expressed with x - i * y. i is an integer.
- If $y \neq 0$, the return value has the same sign as that of *x* and the absolute value is less than *y*.
- If *y* is ± 0 or $x = \pm \infty$, NaN is returned and EDOM is set to errno.
- If *x* is non-numeric or *y* is non-numeric, NaN is returned.
- If *y* is infinite, *x* is returned unless *x* is infinite.

6.13 Diagnostic Function

The following diagnostic function is available.

Function Name	Purpose
__assertfail	Supports the assert macro

__assertfail

Supports the assert macro

[Syntax]

```
#include <assert.h>
int     __assertfail ( char *__msg , char *__cond , char *__file , int __line ) ;
```

[Argument(s)/Return value]

Argument	Return Value
__msg : Pointer to character string to indicate output conversion specification to be passed to printf function *__cond* : Actual argument of assert macro *__file* : Source file name *__line* : Source line number	Undefined

[Description]

- A __assertfail function receives information from assert macro (see 6.3.13 assert.h), calls printf function, outputs information, and calls abort function.
- An assert macro adds diagnostic function to a program.
  When an assert macro is executed, if p is false (equal to 0), an assert macro passes information related to the specific call that has brought the false value (actual argument text, source file name, and source line number are included in the information. The other two are the values of macro __FILE__ and __LINE__, respectively) to __assertfail function.

6.14 Library Stack Consumption List

This section explains the number of stacks consumed for each function in the libraries.

6.14.1 Standard libraries

The number of stacks consumed for each standard library stack function is displayed in the tables below.

(1) ctype.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
isalpha	0	0
isupper	0	0
islower	0	0
isdigit	0	0
isalnum	0	0
isxdigit	0	0
isspace	0	0
ispunct	0	0
isprint	0	0
isgraph	0	0
iscntrl	0	0
isascii	0	0
toupper	0	0
tolower	0	0
toascii	0	0
_toupper	0	0
toup	0	0
_tolower	0	0
tolow	0	0

(2) setjmp.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
setjmp	4	4
longjmp	2	2

(3) stdarg.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
va_arg	0	0
va_start	0	0
va_starttop	0	0
va_end	0	0

(4) stdio.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
sprintf	58 (130)[Note 1]	58 (140)[Note 1]
sscanf	294 (332)[Note 1] (350)[Note 2]	294 (340)[Note 1] (358)[Note 2]
printf	70 (128)[Note 1]	74 (138)[Note 1]
scanf	308 (330)[Note 1] (348)[Note 2]	312 (338)[Note 1] (356)[Note 2]
vprintf	70 (128)[Note 1]	76 (140)[Note 1]
vsprintf	58 (130)[Note 1]	58 (140)[Note 1]
getchar	0	0
gets	8	14
putchar	0	0
puts	6	10
__putc	4	4

Notes 1. Values in parentheses are for when the version that supports floating-point numbers is used.

2. Values in parentheses are for when an operation exception occurs in the version that supports floating-point numbers.

(5) stdlib.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
atoi	4	4
atol	10	10
strtol	20	20
strtoul	20	20
calloc	12	12
free	8	8
malloc	6	6
realloc	12	12
abort	0	0
atexit	0	0
exit	6 + n[Note 1]	6 + n[Note 1]

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
abs	0	0
labs	0	0
div	6 (2)Note 2	6 (2)Note 2
ldiv	18 (8)Note 2	18 (8)Note 2
brk	0	0
sbrk	2	2
atof	46 (64)Note 3	46 (64)Note 3
strtod	46 (64)Note 3	48 (66)Note 3
itoa	10	10
ltoa	16	16
ultoa	16	16
rand	18 (14)Note 4	18 (14)Note 4
srand	0	0
bsearch	36 + nNote 5	40 + nNote 5
qsort	16 + nNote 6	18 + nNote 6
strbrk	0	0
strsbrk	2	2
stritoa	10	10
strltoa	16	16
strultoa	16	16

Notes 1. n is the total stack consumption among external functions registered by the atexit function.

2. Values in the parentheses are for when a multiplier/divider is used.

3. Values in parentheses are for when an operation exception occurs in the version that supports floating-point numbers.

4. Values in the parentheses are for when a multiplier, multiplier/divider is used.

5. n is the stack consumption of external functions called from bsearch.

6. n is (X+ (stack consumption of external functions called from qsort)) - (1 + (number of recursive calls)).

When using a library shared by small model and medium model : X = 38

When using a library shared by large model : X = 40

(6) string.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
memcpy	4	8
memmove	4	6
strcpy	2	6
strncpy	4	10
strcat	2	6
strncat	4	8
memcmp	2	4

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
strcmp	2	2
strncmp	2	2
memchr	2	4
strchr	4	2
strrchr	4	6
strspn	4	6
strcspn	4	4
strpbrk	4	6
strstr	4	8
strtok	4	4
memset	4	6
strerror	0	0
strlen	0	0
strcoll	2	2
strxfrm	4	4

(7) math.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
acos	30 (48)Note	30 (48)Note
asin	30 (48)Note	30 (48)Note
atan	30 (48)Note	30 (48)Note
atan2	30 (48)Note	30 (48)Note
cos	28 (46)Note	28 (46)Note
sin	28 (46)Note	28 (46)Note
tan	34 (52)Note	34 (52)Note
cosh	34 (52)Note	34 (52)Note
sinh	34 (52)Note	34 (52)Note
tanh	40 (58)Note	40 (58)Note
exp	30 (48)Note	30 (48)Note
frexp	2 (16)Note	4 (16)Note
ldexp	0 (16)Note	0 (16)Note
log	30 (48)Note	30 (48)Note
log10	30 (48)Note	30 (48)Note
modf	2 (16)Note	4 (16)Note
pow	30 (48)Note	30 (48)Note
sqrt	22 (40)Note	22 (40)Note
ceil	2 (16)Note	2 (16)Note
fabs	0	0

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
floor	2 (16)[Note]	2 (16)[Note]
fmod	2 (16)[Note]	2 (16)[Note]
matherr	0	0
acosf	30 (48)[Note]	30 (48)[Note]
asinf	30 (48)[Note]	30 (48)[Note]
atanf	30 (48)[Note]	30 (48)[Note]
atan2f	30 (48)[Note]	30 (48)[Note]
cosf	28 (46)[Note]	28 (46)[Note]
sinf	28 (46)[Note]	28 (46)[Note]
tanf	34 (52)[Note]	34 (52)[Note]
coshf	34 (52)[Note]	34 (52)[Note]
sinhf	34 (52)[Note]	34 (52)[Note]
tanhf	40 (58)[Note]	40 (58)[Note]
expf	30 (48)[Note]	30 (48)[Note]
frexpf	2 (16)[Note]	4 (16)[Note]
ldexpf	0 (16)[Note]	0 (16)[Note]
logf	30 (48)[Note]	30 (48)[Note]
log10f	30 (48)[Note]	30 (48)[Note]
modff	2 (16)[Note]	4 (16)[Note]
powf	30 (48)[Note]	30 (48)[Note]
sqrtf	22 (40)[Note]	22 (40)[Note]
ceilf	2 (16)[Note]	2 (16)[Note]
fabsf	0	0
floorf	2 (16)[Note]	2 (16)[Note]
fmodf	2 (16)[Note]	2 (16)[Note]

Note Values in parentheses are for when an operation exception occurs.

(8) assert.h

Function Name	Shared by Small Model and Medium Model	Shared by Large Model
__assertfail	82 (140)[Note]	92 (156)[Note]

Note Values in parentheses are for when the printf version that supports floating-point numbers is used.

6.14.2 Runtime libraries

The number of stacks consumed for each runtime library function is shown in the tables below.

(1) Increment

Function Name	Stack Consumption
lsinc	0
luinc	0
finc	16 (34)[Note]

Note Values in parentheses are for when an operation exception occurs.

(2) Decrement

Function Name	Stack Consumption
lsdec	0
ludec	0
fdec	16 (34)[Note]

Note Values in parentheses are for when an operation exception occurs.

(3) Sign reverse

Function Name	Stack Consumption
lsrev	2
lurev	2
frev	0

(4) 1's complement

Function Name	Stack Consumption
lscom	0
lucom	0

(5) Logical NOT

Function Name	Stack Consumption
lsnot	0
lunot	0

(6) Multiply

Function Name	Stack Consumption
csmul	0
cumul	0
iumul	4 (2)[Note 1]

Function Name	Stack Consumption
lsmul	8 (4)[Note 1]
lumul	8 (4)[Note 1]
fmul	8 (26)[Note 2]

Notes 1. Values in the parentheses are for when a multiplier, multiplier/divider is used.

2. Values in parentheses are for when an operation exception occurs .

(7) Divide

Function Name	Stack Consumption
csdiv	8 (10)[Note 1]
cudiv	2 (4)[Note 1]
isdiv	12 (8)[Note 1]
iudiv	6 (2)[Note 1]
lsdiv	12 (8) [Note 1]
ludiv	6 (2)[Note 1]
fdiv	8 (26)[Note 2]

Notes 1. Values in the parentheses are for when a multiplier/divider is used.

2. Values in parentheses are for when an operation exception occurs .

(8) Remainder

Function Name	Stack Consumption
csrem	8 (10)[Note]
curem	2 (4)[Note]
isrem	12 (8)[Note]
iurem	6 (2)[Note]
lsrem	12 (8)[Note]
lurem	6 (12) [Note]

Note Values in the parentheses are for when a multiplier/divider is used.

(9) Add

Function Name	Stack Consumption
lsadd	0
luadd	0
fadd	8 (26)[Note]

Note Values in parentheses are for when an operation exception occurs.

(10) Subtract

Function Name	Stack Consumption
lssub	2
lusub	2
fsub	8 (26)[Note]

Note Values in parentheses are for when an operation exception occurs.

(11) Shift left

Function Name	Stack Consumption
lslsh	4
lulsh	4

(12) Shift right

Function Name	Stack Consumption
lsrsh	4
lursh	4

(13) Compare

Function Name	Stack Consumption
cscmp	0
iscmp	0
lscmp	2
lucmp	2
fcmp	4 (24)[Note]

Note Values in parentheses are for when an operation exception occurs.

(14) Bit AND

Function Name	Stack Consumption
lsband	0
luband	0

(15) Bit OR

Function Name	Stack Consumption
lsbor	0
lubor	0

(16) Bit XOR

Function Name	Stack Consumption
lsbxor	0
lubxor	0

(17) Conversion from floating point number

Function Name	Stack Consumption
ftols	6
ftolu	6

(18) Conversion to floating point number

Function Name	Stack Consumption
lstof	6
lutof	6

(19) Conversion from bit

Function Name	Stack Consumption
btol	0

(20) Startup routine

Function Name	Stack Consumption
cstart	4

(21) Pre- and post-processing of function

Function Name	Stack Consumption
cpre3e	Size of base pointer + first argument + register variables + automatic variables
cprep3	Size of base pointer + first argument + register variables + automatic variables
cdis3e	0
cdisp3	0
hdwinit	0

(22) BCD-type conversion

Function Name	Purpose
bcdtob	6
btobcd	6
bcdtow	6
wtobcd	8
bbcd	6

(23) Auxiliary

Function Name	Purpose
bcdtob	6
btobcd	6
bcdtow	6
wtobcd	8
bbcd	6

Function Name	Stack Consumption
divuw	6 (2)[Note]
df1in	0
dn4in	0
dn4ip	4
df4in	0
df4ip	4
dn4ino	0
dn4ipo	4
df4ino	0
df4ipo	4
df1de	0
dn4de	0
dn4dp	4
df4de	0
df4dp	4
dn4deo	0
dn4dpo	4
df4deo	0
df4dpo	4
indao	0
ifdao	0
inado	0
ifado	0
lnd0	2
lfd0	2
ln0d	0
lf0d	0
lnd0o	2
lfd0o	2
ln0do	0
lf0do	0

Note Values in the parentheses are for when a multiplier/divider is used.

6.15 List of Maximum Interrupt Disabled Times for Libraries

For libraries that use a multiplier, multiplier/divider, a period of time during which an interrupt is disabled is provided in order that the contents of the operation are not destroyed during an interrupt.

The maximum interrupt disabled times for libraries that use a multiplier, multiplier/divider are shown below.

No periods during which an interrupt is disabled are provided for libraries that do not use a multiplier, multiplier/divider.

Table 6-2. Maximum Interrupt Disabled Time (Number of Clocks) for Libraries

Classification	Function Name	Maximum Interrupt Disabled Time		Remark
		When a multiplier is used	When a multiplier/ divider is used	
Multiplication	@@iumul	12	12	Performs multiplication between unsigned int data
	@@lumul	24	24	Performs multiplication between unsigned long data
	@@lsmul	24	24	Performs multiplication between signed long data
Division	@@cudiv	-	40	Performs division between unsigned char data
	@@csdiv	-	40	Performs division between signed char data
	@@iudiv	-	39	Performs division between unsigned int data
	@@isdiv	-	39	Performs division between signed int data
	@@ludiv	-	43	Performs division between unsigned long data
	@@lsdiv	-	43	Performs division between signed long data
Remainder arithmetic	@@curem	-	40	Performs remainder arithmetic between unsigned char data
	@@csrem	-	40	Performs remainder arithmetic between signed char data
	@@iurem	-	39	Performs remainder arithmetic between unsigned int data
	@@isrem	-	39	Performs remainder arithmetic between signed int data
	@@lurem	-	43	Performs remainder arithmetic between unsigned long data
	@@lsrem	-	43	Performs remainder arithmetic between signed long data
Auxiliary	@@divuw	—	43	divuw instruction compatibility
stdio.h	printf	—	43Note	Outputs data to SFR
	sprintf	—	43Note	Writes data to a string
	vprintf	—	43Note	Outputs data to SFR
	vsprintf	—	43Note	Writes data to a string

Classification	Function Name	Maximum Interrupt Disabled Time		Remark
		When a multiplier is used	When a multiplier/ divider is used	
stdlib.h	div	—	41	Performs int type division
	ldiv	—	46	Performs long type division
	rand	24	24	Uses @@lumul
	qsort	12	12	Uses @@iumul

Note Values in parentheses are for when the version that supports floating-point numbers is used.

6.16 Batch Files for Update of Startup Routine and Library Functions

The 78K0R C compiler provides batch files for updating a portion of the standard library functions and the startup routine. The batch files in the bat folder are shown in the table below.

Table 6-3. Batch Files for Updating Library Functions

Batch File	Application
mkstup.bat	Updates the startup routine (cstart*.asm). When changing the startup routine, perform assembly using this batch file.
reprom.bat	Updates the firmware ROMization termination routine (rom.asm). When changing rom.asm, update the library using this batch file.
repgetc.bat	Updates the getchar function. The default assumption sets P0 of the SFR to the input port. When it is necessary to change this setting, change the defined value of EQU of PORT in getchar.asm and update the library using this batch file.
repputc.bat	Updates the putchar function. The default assumption sets P0 of the SFR to the output port. When it is necessary to change this setting, change the defined value of EQU of PORT in getchar.asm and update the library using this batch file.
repputcs.bat	Updates the putchar function to SM+ for 78K0R for C compiler-compatibility. When it is necessary to check the output of the putchar function using the SM+ for 78K0R for C compiler, update the library using this batch file.
repselo.bat	Saves/restores the reserved area of the compiler (_@KREGxx) as part of the save/restore processing of the setjmp and longjmp functions (the default assumption is to not save/restore). Update the library using this batch file when the -qr option is specified.
repselon.bat	Does not save/restore the reserved area of the compiler (_@KREGxx) as part of the save/ restore processing of the setjmp and longjmp functions (the default assumption is to not save/ restore). Update the library using this batch file when the -qr option is not specified.
repvect.bat	Updates the address value setting processing of the branch table of the interrupt vector table allocated in the flash area (vect*.asm). The default assumption sets the top address of the flash area branch table to 2000H. When it is necessary to change this setting, change the defined value of EQU of ITBLTOP in vect.inc and update the library using this batch file.
repmul.bat	Updates the multiplier library.
repmuldiv.bat	Updates the multiplier/divider library.

6.16.1 Using batch files

Use the batch files in the subfolder bat.

Because these files are the batch files used to activate the assembler and librarian, the assembler etc. bundled to CubeSuite are necessary. Before using the batch files, set the folder that contains the 78K0R assembler execution format file using the environment variable PATH.

Create a subfolder (lib) of the same level as bat for the batch files and put the post-assembly files in this subfolder. When a C startup routine or library is installed in a subfolder lib that is the same level as bat, these files are overwritten

Files assembled with the batch files are output to Src\cc78k0\lib. Copy these files to the lib78k0 directory before linking.

To use the batch files, move the current folder to the subfolder bat and execute each batch file. To perform execution, the following parameters are necessary.

Product type = chiptype (classification of target chip)

f1166a0 : uPD78F1166_A0etc.

The following is an illustration of how to use each batch file.

(1) Startup routine

```
mkstup chiptype
```

Example below.

```
mkstup f1166a0
```

(2) Firmware ROMization routine update

```
reprom chiptype
```

Example below.

```
reprom f1166a0
```

(3) getchar function updat

```
regetc chiptype
```

Example below.

```
repgetc f1166a0
```

(4) putcharfunction update

```
repputc chiptype
```

Example below.

```
repputc f1166a0
```

(5) putchar function (SM78K0R-supporting) update

```
repputcs chiptype
```

Example below.

```
repputcs f1166a0
```

(6) setjmp/longjmp function update (with restore/save processing)

```
repselo chiptype
```

Example below.

```
repselo f1166a0
```

(7) setjmp/longjmp function update (without restore/save processing)

```
repselon chiptype
```

Example below.

```
repselon f1166a0
```

(8) Interrupt vector table update

```
repvect chiptype
```

Example below.

```
repvect f1166a0
```

(9) Multiplier use library update

```
repmul.bat chiptype
```

The example updated for UPD78F1235_64 is shown below.

```
repmul.bat f123564
```

Below is updated.

src\cc78k0r\lib\cl0rxm.lib
cl0rxme.lib
cl0rxl.lib
cl0rxle.lib

(10) Multiplier/divider use library update

```
repmuldiv.bat chiptype
```

The example updated for UPD78F1235_64 用 is shown below.

```
repmuldiv.bat f123564
```

Below is updated.

src\cc78k0r\lib\cl0rdm.lib
cl0rdme.lib
cl0rdl.lib
cl0rdle.lib

CHAPTER 7 STARTUP

This chapter explains the startup routine.

7.1 Function Overview

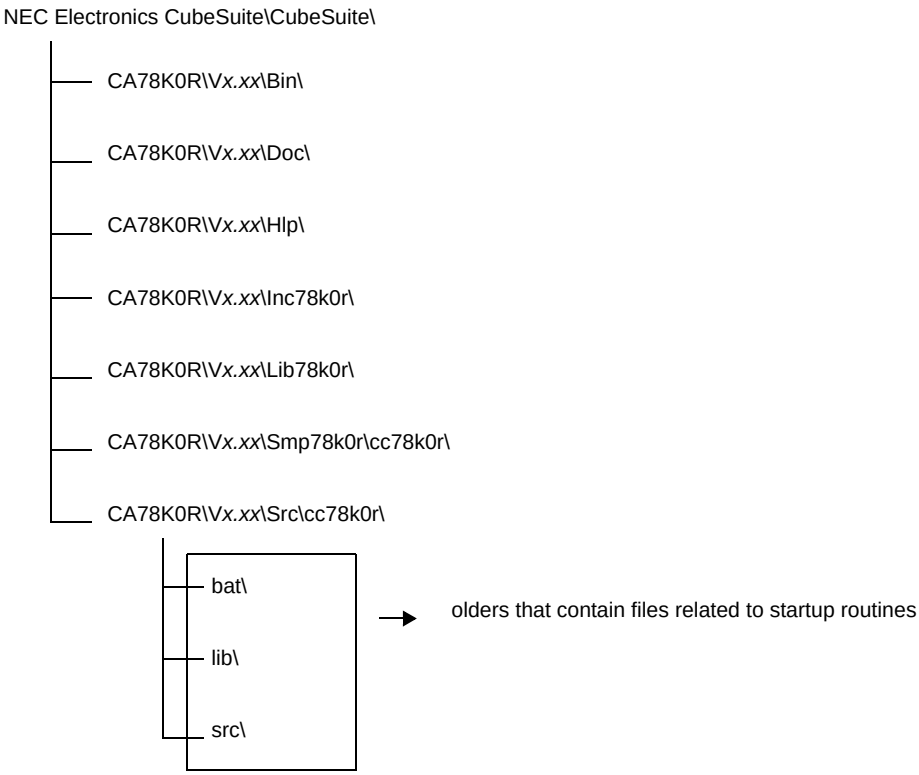
To execute a C language program, a program is needed to handle ROMization for inclusion in the system and to start the user program (main function). This program is called the startup routine.

To execute a user program, a startup routine must be created for that program. The CA78K0R provides standard startup routine object files, which carry out the processing required before program execution, and the startup routine source files (assembly source), which the user can adapt to the system. By linking the startup routine object file to the user program, an executable program can be created without requiring the user to write an original execution preprocessing routine.

This chapter describes the contents and uses of the startup routine and explains how to adapt it for your system.

7.2 File Organization

The files related to a startup routine are stored in the folder Src\cc78k0r of the C compiler package.



The contents of the folders under Src\cc78k0r are shown next.

7.2.1 "bat" folder contents

Batch file in this folder cannot be used in the IDE.

Use these batch files only when a source file, such as for the startup routine, must be modified.

Table 7-1. "bat" Folder Contents

Batch File Name	Explanation
mkstup.bat	Assemble batch file for startup routine
reprom.bat	Batch file for updating rom.asm[Note 1]
repgetc.bat	Batch file for updating getchar.asm
repputc.bat	Batch file for updating putchar.asm
repputcs.bat	Batch file for updating _putchar.asm
repselo.bat	Batch file for updating setjmp.asm and longjmp.asm (the compiler reserved area is saved) [Note 2]
repselon.bat	Batch file for updating setjmp.asm, longjmp.asm (the compiler reserved area is not saved))[Note 2]
repvect.bat	Batch file for updating vect*.asm
repmul.bat	Batch file for updating multiplier library
repmuldiv.bat	Batch file for updating multiplier/divider library

Notes 1. Since ROMization routines are in the library, the library is also updated by this batch file.

2. setjmp and longjmp functions that save the compiler reserved area (saddr area secured for KREGxx, etc.), and setjmp and longjmp functions that do not save the compiler reserved area (only registers are saved) are created.

7.2.2 "lib" folder contents

The lib folder contains the object files that were assembled from the source files of the startup routine and libraries. These object files can be linked with programs for any 78K0R target device. If code modifications are not especially needed, link the default object files as is. The object files are overwritten when batch file mkstup.bat, which is provided by the CA78K0R, is executed.

Table 7-2. "lib" folder Contents

File Name			File Role
Normal	Boot Area	Flash Area	
cl0rm.lib cl0rl.lib cl0rmf.lib cl0rlf.lib cl0rxm.lib cl0rdm.lib cl0rxl.lib cl0rdl.lib	cl0rm.lib cl0rl.lib cl0rmf.lib cl0rlf.lib cl0rxm.lib cl0rdm.lib cl0rxl.lib cl0rdl.lib	cl0rme.lib cl0rle.lib cl0rmfe.lib cl0rlfe.lib cl0rxme.lib cl0rdme.lib cl0rxle.lib cl0rdle.lib	Library (runtime and standard libraries)[Note 1]
s0rm.rel s0rml.rel s0rl.rel s0rll.rel	s0rmb.rel s0rmlb.rel s0rlb.rel s0rllb.rel	s0rme.rel s0rmle.rel s0rle.rel s0rlle.rel	Object files for startup routines[Note 2]

Notes 1. The rule for naming libraries is given below.

```
lib78k0r\cl0r<mul><model><float><flash>.lib
```

<mul>

None : Standard library
x : Multiplier used
d : Multiplier/divider used

<model>

m : Small model or medium model
l : Large model

<float>

None : Standard library and runtime library (floating point library is not used)
f : For floating point library

<flash>

None : For normal/boot area
e : For flash memory area

2. The rule for naming startup routines is given below.

```
lib78k0r\s0r<model><lib><flash>.rel
```

<model>

m : Medium model (can also be used for specifying the small model)
l : Large model

<lib>

None : When standard library functions are not used
l : When standard library functions are used

<flash>

None : Normal
b : For boot area
e : For flash memory area

3. The 78K0R C compiler libraries are compatible with the following multiplier, multiplier/divider devices. However, if an interrupt occurs while computation is in progress, some of the computation reslts are disabled from being interrupted so that they are not corrupted.
See "6.15 List of Maximum Interrupt Disabled Times for Libraries" regarding library functions and interrupt disable times.
The special function registers is given below.

Function	Reserved Words	Adresses	Size
Multiplication input data A	MULA	FFFF0H	16bit
Multiplication input data B	MULB	FFFF2H	16bit
Multiplication result data	MULOH, MULOL	FFFF4H, FFF6H	16bit x 2

- Register configuration

< Multiplier A > < Multiplier B > < Product >

MULA (bits 15 to 0) x MULB (bits 15 to 0) = MULOH (upper) (bits 15 to 0), MULOL (lower) (bits 15 to 0)

7.2.3 "src" folder contents

The src folder contains the assembler source files of the startup routines, ROM routines, error processing routines, and standard library functions (a portion). If the source must be modified to conform to the system, the object files for linking can be created by modifying this assembler source and using a batch file in the bat folder to assemble.

Table 7-3. "src" Folder Contents

Startup Routine Source File Name	Explanation
cstart.asm[Note]	Source file for startup routine (when standard library is used)
cstartn.asm[Note]	Source file for startup routine (when standard library is not used)
rom.asm	Source file for ROMization routine
_putchar.asm	_putchar function
putchar.asm	putchar function
getchar.asm	getchar function
longjmp.asm	longjmp function
setjmp.asm	setjmp function
vectxx.asm	Vector source for each interrupt (xx : vector address)
def.inc	For setting library according to type
macro.inc	Macro definition for each typical pattern
vect.inc	Start address of flash memory area branch table
library.inc	Selection of library assigned to boot area explicitly
imul.asm, lmul.asm	Function for multiplier, multiplier/divider libraries
csdiv.asm, cudiv.asm, csrem.asm, curem.asm, isdiv.asm, iudiv.asm, isrem.asm, iurem.asm, lsdiv.asm, ludiv.asm, lsrem.asm, lurem.asm, divuw.asm, div.asm, ldiv.asm	Function for multiplier/divider libraries

Note A file name with "n" added is a startup routine that does not have standard library processing. Use only if the standard library will not be used. Additionally, boot area startup routines are named cstartb*.asm, and flash area startup routines are named cstarte*.asm.

7.3 Batch File Description

This section explains batch file.

7.3.1 Batch files for creating startup routines

The mkstup.bat in the bat folder is used to create the object file of a startup routine.

The assembler in the CA78K0R is required for mkstup.bat. Therefore, if PATH is not specified, specify it before running the batch file.

How to use this file is described next.

- Execute the following command line in the Src\cc78k0r\bat folder containing mkstup.bat.

```
mkstup device-type[Note]
```

Note See the user's manual of the target device or "Device File Operating Precautions".

An example of use is described next.

- This example creates a startup routine to use when the target device is the uPD78F1166_A0.

```
mkstup f1166a0
```

The mkstup.bat batch file stores the new startup routine so as to overwrite the object files of the startup routine in the lib folder at the same level as the bat folder.
The startup routine that is required to link object files is output to each folder.
The names of the object files created in lib are shown below.

lib —— s0rm.rel
s0rmb.rel
s0rme.rel
s0rml.rel
s0rmlb.rel
s0rmle.rel
s0rl.rel
s0rlb.rel
s0rle.rel
s0rll.rel
s0rllb.rel
s0rlle.rel

7.4 Startup Routines

This section explains startup routines.

7.4.1 Overview of startup routines

A startup routine makes the preparations needed to execute the C source program written by the user. By linking it to a user program, a load module file that achieves the objective of the program can be created.

(1) Function

Memory initialization, ROMization for inclusion in a system, and the entry and exit processes for the C source program are performed.

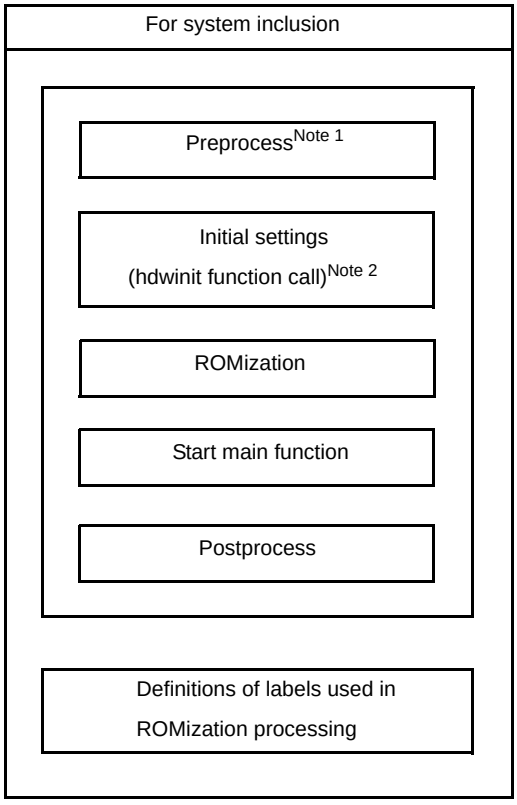
- ROMization

  The initial values of the external variables, static variables, and sreg variables defined in the C source program are located in ROM. However, the variable values cannot be rewritten; only placed in ROM as is. Therefore, the initial values located in ROM must be copied to RAM. This process is called ROMization. When a program is written to ROM, it can be run by a microcontroller.

(2) Configuration

The figure below shows the programs related to the startup routines and their configurations.

Figure 7-1. Programs Related to Startup Routines and Their Configurations



Notes 1. If the standard library is used, the processing related to the library is performed first. Startup routine source files that do not have an "n" appended at the end of their file names are processed in relation to the standard library. Files with the appended "n" are not processed.

2. The hdwinit function is a function created as necessary by the user as a function to initialize peripheral devices (sfr). By creating the hdwinit function, the timing of the initial settings can be accelerated (the initial settings can be made in the main function). If the user does not create the hdwinit function, the process returns without doing anything.

cstart.asm and cstartn.asm have nearly identical contents.
The table below shows the differences between cstart.asm and cstartn.asm.

Type of Startup Routine	Uses Library Processing
cstart.asm	Yes
cstartn.asm	No

(3) Uses of startup routines

The table below lists the names of the object files for the source files provided by the CA78K0R.

File Type	Source File	Object File
Startup routine	cstart*.asm Note 1, 2	s0r*.rel Note 2, 3, 4
ROMization file	rom.asm	Included in library

Notes 1. *: If the standard library is not used, "n" is added. If the standard library is used, "n" is not added.

2. *: "b" is added for boot area startup routines, and "e" for flash area startup routines.

3. *: If a fixed area in the standard library is used, "l" is added.

4. *: If the small model or medium model is specified, "m" is added. If the large model is specified, "l" is added.
   Even when using the small model or medium model, use the startup routine to which "l" is added if variables are allocated in the far area.

Remark rom.asm defines the label indicating the final address of the data copied by ROMization.
The object file generated from rom.asm is included in the library.

7.4.2 Startup routine preprocessing

Sample program (cstart.asm) preprocessing will now be explained.

Remark cstart is called in the format with _@ added to its head.

```
        NAME    @cstart

$INCLUDE ( def.inc )                                                    ; (1)
$INCLUDE ( macro.inc )
                                                                        ; (2)
BRKSW     EQU   1     ; brk, sbrk, calloc, free, malloc, realloc function use
EXITSW    EQU   1     ; exit, atexit function use
RANDSW    EQU   1     ; rand, srand  function use
DIVSW     EQU   1     ; div           function use
LDIVSW    EQU   1     ; 1div          function use
FLOATSW   EQU   1     ; floating point variables use
STRTOKSW  EQU   1     ; strtok        function use

        PUBLIC  _@cstart, _@cend                                        ; (3)

$_IF ( BRKSW )
        PUBLIC  _@BRKADR, _@MEMTOP, _@MEMBTM
          :
$ENDIF
        EXTRN   _main, _@STBEG, _hdwinit, _@MAA                         ; (4)
$_IF ( EXITSW )
        EXTRN   _exit
$ENDIF
                                                                        ; (5)
        EXTRN   _?R_INIT, _?RLINIT, _?R_INIS, _?DATA, _?DATAL, _?DATS
@@DATA  DSEG    BASEP ; near                                            ; (6)

$_IF ( EXITSW )
_@FNCTBL :      DS    4 * 32
_@FNCENT :      DS    2
  :
_@MEMTOP :      DS    32
_@MEMBTM :
$ENDIF
```

(1) Including include files

def.inc -> For settings according to the library type.
macro.inc -> Macro definitions for typical patterns.

(2) Library switch

If the standard libraries listed in the comments are not used, by changing the EQU definition to 0, the space secured for the processing of unused libraries and for use by the library can be conserved. The default is set to use everything (In a startup routine without library processing, this processing is not performed).

(3) Symbol definitions

The symbols used when using the standard library are defined.

(4) External reference declaration of symbol for stack resolution

The public symbol (_@STBEG) for stack resolution is an external reference declaration.
_@STBEG has the value of the last address in the stack area + 1.
_@STBEG is automatically generated by specifying the symbol generation option (-s) for stack resolution in the linker. Therefore, always specify the -s option when linking. In this case, specify the name of the area used in the stack. If the name of the area is omitted, the RAM area is used, but the stack area can be located anywhere by creating a link directive file. For memory mapping, see the user's manual of the target device.
An example of a link directive file is shown below. The link directive file is a text file created by the user in an ordinary editor (for details about the description method, see "7.6 Coding Examples").

Example When -sSTACK is specified in linking
Create lk78k0r.dr (link directive file). Since ROM and RAM are allocated by default operations by referencing the memory map of the target device, it is not necessary to specify ROM and RAM allocations unless they need to be changed.
For link directives, see lk78k0r.dr in the Smp78k0r\CA78K0R folder.

```
                First address  Size
                      |         |
memory  SDR     : ( 0xFFE20h, 0000098h )
memory  STACK   : ( 0xxxxxxh, 0xxxxxxh ) <- Specify the first address and size here,
                                            then specify lk78k0r.dr with the -d
                                            linker option.78k0r
                                            (Example: -dlk78k0r.dr)
merge @@INIS    : = SDR
merge @@DATS    : = SDR
merge @@BITS    : = SDR
```

(5) External reference declaration of labels for ROMization processing

The labels for ROMization processing are defined in the postprocessing section.

(6) Securing area for standard library

The area used when using the standard library is secured.

7.4.3 Startup routine initial settings

The initial settings for a sample program (cstart.asm) will now be explained.

```
@@VECT00        CSEG    AT      0                                       ; (1)
        DW      _@cstart

@@LCODE CSEG    BASE
_@cstart :
        SEL     RB0                                                     ; (3)
        MOV     A, #_@MAA                                               ; (2)
        MOV1    CY, A.0
        MOV1    MAA, CY
        MOVW    SP, #LOWW _@STBEG       ; SP <-stack begin address      ; (4)
        CALL    !!_hdwinit                                              ; (5)
          :
$_IF ( BRKSW OR EXITSW OR RANDSW OR FLOATSW )
        CLRW    AX
$ENDIF
          :
```

(1) Reset vector setting

The segment of the reset vector table is defined as follows. The first entry address of the startup routine is set.

```
@@VECT00        CSEG    AT      0000H
                DW      _@cstart
```

(2) Mirror area setting

The mirror area is set.

Regarding the mirror area, refer to the user's manual of the target device.

(3) Register bank setting

Register bank RB0 is set as the work register bank.

(4) Stack pointer (SP) setting

_@STBEG is set in the stack pointer.

_@STBEG is automatically generated by specifying the symbol generation option (-s) for stack resolution in the linker.

(5) Hardware initialization function call

The hdwinit function is created when needed by the user as the function for initializing peripheral devices (SFR).

By creating this function, initial settings can be made to match the user's objectives.

If the user does not create the hdwinit function, the process returns without doing anything.

(6) ROMization processing

The ROMization processing in cstart.asm will now be described.

```
; copy external variables having initial value
$_IF ( _ESCOPY )
        MOV     ES, #HIGHW _@R_INIT
$ENDIF
        MOVW    HL, #LOWW _@R_INIT
        MOVW    DE, #LOWW _@INIT
        BR      $LINIT2
LINIT1 :
$_IF ( _ESCOPY )
        MOV     A, ES : [HL]
$ELSE
        MOV     A, [HL]
$ENDIF
        MOV     [DE], A
        INCW    HL
        INCW    DE
LINIT2 :
        MOVW    AX, HL
        CMPW    AX, #LOWW _?R_INIT
        BNZ     $LINIT1
```

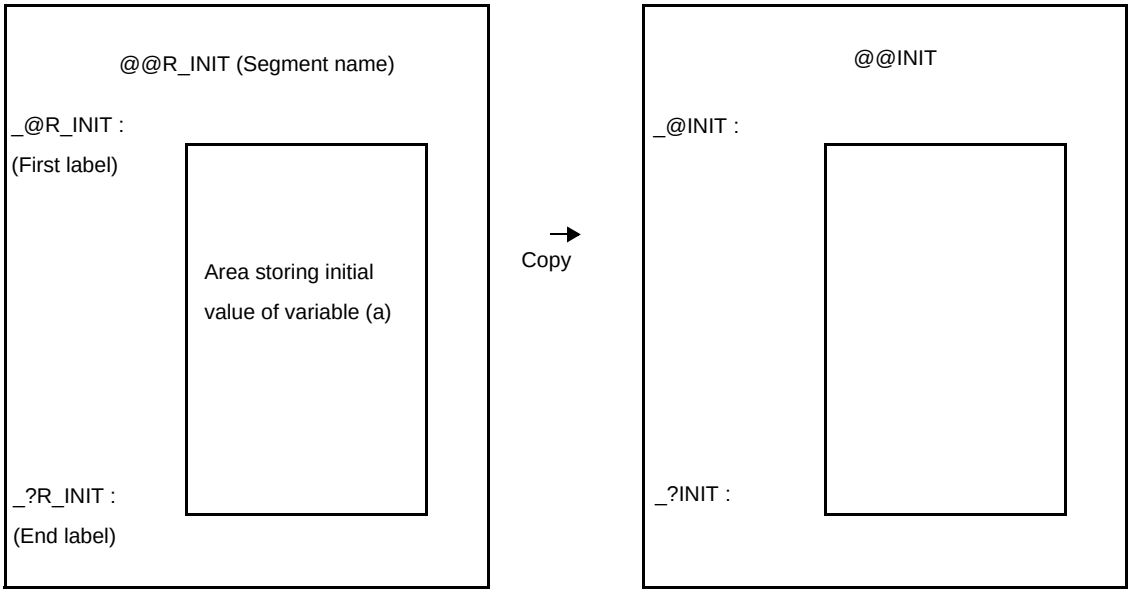
In ROMization processing, the initial values of the external variables and the sreg variables stored in ROM are copied to RAM. The variables to be processed have the 4 types (a) to (d) shown in the following example.

```
char    c = 1 ;                 (a)External variable with initial value
int     i ;                     (b)External variable without initial value[Note]
__sreg  int     si = 0 ;        (c)sreg variable with initial value
__sreg  char    sc ;            (d)sreg variable without initial value[Note]

void main ( void ) {
          :
}
```

Note External variables without initial value and sreg variables without initial value are not copied, and zeros are written directly to RAM.

- The figure below shows ROMization processing for external variable (a) with an initial value.
  The initial value of the variable (a) is placed in the @@R_INIT segment of ROM by the 78K0R C compiler.
  The ROMization processing copies this value to the @@INIT segment in RAM (the same processes are performed for the variable (c)).

Figure 7-2. ROMization Processing for External Variable with Initial Value



- The first and end labels in the @@R_INIT segment are defined by _@R_INIT and _?R_INIT. The first and end labels in the @@INIT segment are defined by _@INIT and _?INIT.
- The variables (b) and (d) are not copied, but zeros are written directly in the predetermined segment in RAM.

The table below shows the segment names of the ROM areas where the variables (a) to (c) are placed, and the first and end labels of the initial values in each segment.

Variable Type	Segment	First Label	End Label
External variable with initial value (a) (when allocated to near area)	@@R_INIT	_@R_INIT	_?R_INIT
External variable with initial value (a) (when allocated to far area)	@@RLINIT	_@RLINIT	_?RLINIT
sreg variable with initial value(c)	@@R_INIS	_@R_INIS	_?R_INIS

The table below shows the segment names of the RAM areas where the variables (a) to (d) are placed, and the first and end labels of the initial values in each segment.

Variable Type	Segment	First Label	End Label
External variable with initial value (a) (when allocated to near area)	@@INIT	_@INIT	_?INIT
External variable with initial value (a) (when allocated to far area)	@@INITL	_@INITL	_?INITL
External variable without initial value (b) (when allocated to near area)	@@DATA	_@DATA	_?DATA
External variable without initial value (b) (when allocated to far area)	@@DATAL	_@DATAL	_?DATAL
sreg variable with initial value (c)	@@INIS	_@INIS	_?INIS
sreg variable without initial value (d)	@@DATS	_@DATS	_?DATS

7.4.4 Startup routine main function startup and postprocessing

Starting the main function and postprocessing in a sample program (cstart.asm) will now be described.

```
            CALL    !!_main          ; main ( );             ; (1)
$_IF ( EXITSW )
            CLRW    AX
            CALL    !!_exit          ; exit ( 0 );           ; (2)
$ENDIF
            BR      $$
;
_@cend :
                                                            ; (3)
@@R_INIT    CSEG    UNIT64KP
_@R_INIT :
@@RLINIT    CSEG    UNIT64KP
_@RLINIT :
@@R_INIS    CSEG    UNIT64KP
_@R_INIS :
@@INIT      DSEG    BASEP
_@INIT :
@@INITL     DSEG    UNIT64KP
_@INITL :
@@DATA      DSEG    BASEP
_@DATA :
@@DATAL     DSEG    UNIT64KP
_@DATAL :
@@INIS      DSEG    SADDRP
_@INIS :
@@DATS      DSEG    SADDRP
_@DATS :
@@CALT      CSEG    CALLT0
@@CNST      CSEG    MIRRORP
@@CNSTL     CSEG    PAGE64KP
@@BITS      BSEG
;
            END
```

(1) Starting the main function

The main function is called.

(2) Starting the exit function

When exit processing is needed, the exit function is called.

(3) Definitions of segments and labels used in ROMization processing

The segments and labels used for each variable (1) to (4) (see "(6) ROMization processing") in ROMization processing are defined. A segment indicates the area that stores the initial value of each variable. A label indicates the first address in each segment.

The ROMization processing file rom.asm will now be described. The rom.asm relocatable object file is in the library.

```
        NAME    @rom
;
        PUBLIC  _?R_INIT, _?RLINIT, _?R_INIS
        PUBLIC  _?INIT, _?INITL, _?DATA, _?DATAL, _?INIS, _?DATS
;
@@R_INIT        CSEG    UNIT64KP                        ; (4)
_?R_INIT :
@@RLINIT        CSEG    UNIT64KP
_?RLINIT :
@@R_INIS        CSEG    UNIT64KP
_?R_INIS :
@@INIT          DSEG    BASEP
_?INIT :
@@INITL         DSEG    UNIT64KP
_?INITL :
@@DATA          DSEG    BASEP
_?DATA :
@@DATAL         DSEG    UNIT64KP
_?DATAL :
@@INIS          DSEG    SADDRP
_?INIS :
@@DATS          DSEG    SADDRP
_?DATS :
;
        END
```

(4) Definition of labels used in ROMization processing

The labels used for each variable (1) to (4) (see "(6) ROMization processing) in ROMization processing are defined. These labels indicate the last address of the segment storing the initial value of each variable.

If multiple user libraries exist and mutual references exist between the object module files belonging to these libraries, do not change the module names "@rom" and "@rome" of the CA78K0R terminal module.

If the terminal module name is changed, it may not link in the end.

7.5 ROMization Processing in Startup Module for Flash Area

The startup modules for flash differ with the ordinary startup modules as follows.

Table 7-4. ROM Area Section for Initialization Data

Variable Type	Segment	First Label	End Label
External variable with initial value (a) (when allocated to near area)	@ER_INIT CSEG UNIT64KP	E@R_INIT	E?R_INIT
External variable with initial value (a) (when allocated to far area)	@ERLINIT CSEG UNIT64KP	E@RLINIT	E?RLINIT
sreg variable with initial value(c)	@ER_INIS CSEG UNIT64KP	E@R_INIS	E?R_INIS

Table 7-5. RAM Area Section of Copy Destination

Variable Type	Segment	First Label	End Label
External variable with initial value (a) (when allocated to near area)	@EINIT DSEG BASEP	E@INIT	E?INIT
External variable with initial value (a) (when allocated to far area)	@EINITL DSEG UNIT64KP	E@INITL	E?INITL
External variable without initial value (b) (when allocated to near area)	@EDATA DSEG BASEP	E@DATA	E?DATA
External variable without initial value (b) (when allocated to far area)	@EDATAL DSEG UNIT64KP	E@DATAL	E?DATAL
sreg variable with initial value (c)	@EINIS DSEG SADDRP	E@INIS	E?INIS
sreg variable without initial value (d)	@EDATS DSEG SADDRP	E@DATS	E?DATS

- In the startup module, the following labels are added at the head of each segment in ROM area and RAM area.
  E@R_INIT, E@R_INIS, E@INIT, E@DATA, E@INIS, E@DATS, E@INITL, E@DATAL
  Furthermore, the following labels are added if the large model is specified or variables are allocated in the far area.
  E@RLINIT, E@INITL, E@DATAL
- In the terminal module, the following labels are added at the terminal of each segment in ROM area and RAM area.
  E?R_INIT, E?R_INIS, E?INIT, E?DATA, E?INIS, E?DATS, E?RLINIT, E?INITL, E?DATAL
- The startup module copies the contents from the first label address of each segment in ROM area to the end label address -1, to the area from the first label address of each segment in RAM area
- Zeros are embedded from E@DATA to E?DATA, and from E@DATS to E?DATS.
- Furthermore, zeros are embedded from E@DATAL to E?DATAL if the large model is specified or variables are allocated to the far area

7.6 Coding Examples

The startup routines provided by the CA78K0R can be revised to match the target system actually being used. The essential points about revising these files are explained in this section.

7.6.1 When revising startup routine

The essential points about revising a startup routine source file will now be explained. After revising, use the batch file mkstup.bat in the Src\cc78k0r\bat folder to assemble the revised source file (cstart*.asm) (*:alphanumeric symbols).

(1) Symbols used in library functions

If the library functions listed in the table below are not used, the symbols corresponding to each function in the startup routine (cstart.asm) can be deleted. However, since the exit function is used in the startup routine, _@FNCTBL and _@FNCENT cannot be deleted (if the exit function is deleted, these symbols can be deleted). The symbols in the unused library functions can be deleted by changing the corresponding library switch.

Library Function Name	Symbols Used
brk sbrk malloc calloc realloc free	_errno _@MEMTOP _@MEMBTM _@BRKADR

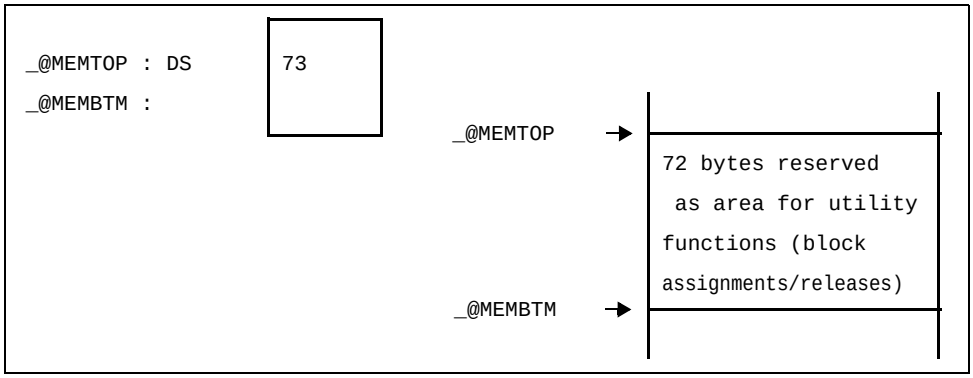
Library Function Name	Symbols Used
exit	_@FNCTBL _@FNCENT
rand srand	_@SEED
div	_@DIVR
ldiv	_@LDIVR
strtok	_@TOKPTR
atof strtod Mathematical function Floating-point runtime library	_errno

(2) Areas that are used for utility functions (block assignments/releases)

If the size of the area used by a utility function (block assignment/release) is defined by the user, this is set as in the following example.

Example If you want to reserve 72 bytes for use by utility functions (block assignments/releases), make the following changes to the initial settings of the startup routine.

Figure 7-3. Startup Routine Initial Settings Example

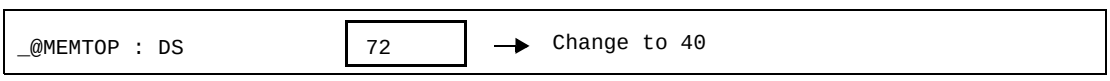


Add one byte to the area size to be secured and then specify the value in the startup routine. In the above example, 73 bytes are secured in the startup routine, but up to 72 bytes can actually be secured for utility functions.

If the specified size is too big to be stored in the RAM area, errors may occur when linking.

In this case, decrease the size specified as shown below, or avoid by correcting the link directive file. For correction of the link directive file, see "5.3.2 When using compiler".

Example To decrease the specified size



7.6.2 When using RTOS

The RX78K0R and 78K0R C compiler provide sample programs for initialization routines (assembler format). When using the RX78K0R and 78K0R C compiler in combination, initialization routines for both tools must therefore be modified.

CHAPTER 8 ROMIZATION

ROMization refers to the process of placing initial values, such as those for initialized external variables, into ROM and then copying them to RAM when the system is executed.

The CA78K0R provides startup routines with built-in program ROMization processing, saving the trouble of writing a routine for romization processing at startup.

For information about the startup routines, see "7.4 Startup Routines".

The method for performing program ROMization is as follows.

During ROMization the startup routine, object module files and libraries are linked. The startup routine initializes the object program.

(1) s0r*.rel

Startup routine (ROMization compatible).

The copy routine for the initialization data is included, and the beginning of the initial data is indicated. The label "_@cstart" (symbol) is added to the start address.

(2) cl0r*.lib

Library included with the CA78K0Rr

The library files of the C compiler include the following library types.

- Runtime library

  "@@" is added to the beginning of the symbol for runtime library names. For special libraries cprep and cdisp, however, "_@" is added to the beginning of the symbol.

- Standard library

  "_" is added to the beginning of the symbol for standard library names.

(3) *.lib

User-created library

"_" is added to the beginning of the symbol.

Caution The CA78K0R provides several types of startup routines and libraries. See "CHAPTER 7 STARTUP" regarding startup routines. See "7.2.2 "lib" folder contents" regarding libraries.

CHAPTER 9 REFERENCING COMPILER AND ASSEMBLER

This chapter describes how to link a program written in assembly language.

If a function called from a C source program is written in another language, both object modules are linked by the linker. This chapter describes the procedure for calling a program written in another language from a program written in the C language, and the procedure for calling a program written in the C language from a program written in another language.

Using the CA78K0R to interface with another language is described in the following order:

- Accessing Arguments and Automatic Variables
- Register HL is used as a base pointer for accessing arguments and automatic variables stored in a stack.Storing Return Values
- Calling Assembly Language Routines from C Language
- Calling C Language Routines from Assembly Language
- Referencing Variables Defined in C Language
- Referencing Variables Defintед in Assembly Language from C Language
- Points of Caution for Calling Between C Language Functions and Assembler Functions

9.1 Accessing Arguments and Automatic Variables

For details on the assignment of argument and automatic variables, see "3.4.2 Ordinary function call interface".

9.2 Register HL is used as a base pointer for accessing arguments and automatic variables stored in a stack.Storing Return Values

See "3.4.1 Return values".

9.3 Calling Assembly Language Routines from C Language

This section shows examples ofdefault procedures.

Calling an assembly language routine from the C language is described as follows.

- C language function calling procedure
- Saving data from assembly language routine and returning

9.3.1 C language function calling procedure

The following is a C language program example that calls an assembly language routine.

```
extern  int     FUNC ( int, long );     /* Function prototype */

void main ( void ) {
        int     i, j ;
        long    l ;

        l = 0x54321 ;
        i = 1 ;
        j = FUNC ( i, l );              /* Function call */
}
```

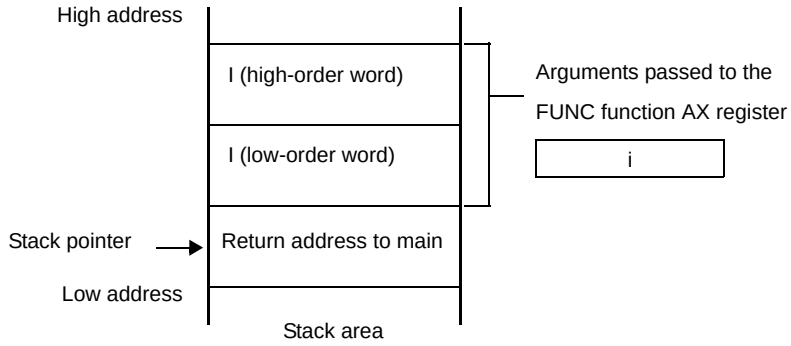
In this program example, the interface and control flow with the program that is executing are as follows.

(a) Placing the first argument passed from the main function to the FUNC function in the register, and the second and subsequent arguments on the stack.

(b) Passing control to the FUNC function by using the CALL instruction.

The next figure shows the stack immediately after control moves to the FUNC function in the above program example.

Figure 9-1. Stack Immediately After Function Is Called



9.3.2 Saving data from assembly language routine and returning

The following processing steps are performed in the FUNC function called by the main function.

(1) Save the base pointer, saddr area for register variable.

(2) Copy the stack pointer (SP) to the base pointer (HL).

(3) Perform the processing in the FUNC function.

(4) Set the return value.

(5) Restore the saved register.

(6) Return to the main function

An example of an assembly language program is explained next.

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _FUNC
        PUBLIC  _DT1
        PUBLIC  _DT2

@@DATA  DSEG    BASEP
_DT1 :  DS      ( 2 )
_DT2 :  DS      ( 4 )

@@CODE  CSEG
_FUNC :
        PUSH    HL              ; save base pointer     (1)
        PUSH    AX
        MOVW    HL, SP          ; copy stack pointer    (2)
        MOVW    AX, [HL]        ; arg1
        MOVW    !_DT1, AX       ; move 1st argument ( i )
        MOVW    AX, [HL + 10]   ; arg2
        MOVW    !_DT2 + 2, AX
        MOVW    AX, [HL + 8]    ; arg2
        MOVW    !_DT2, AX       ; move 2nd argument ( l )
        MOVW    BC, #0AH        ; set return value      (4)
        POP     AX
        POP     HL              ; restore base pointer  (5)
        RET                     ;                       (6)
        END
```

(1) Saving base pointer and work register

A label with "_" is prefixed to the function name written in the C source. Base pointers and work registers are saved with the same name as function names written inside the C source.

After the label is specified, the HL register (base pointer) is saved.

In the case of programs generated by the C compiler, other functions are called without saving the saddr area for register variables. Therefore, if changing the values of these registers for functions that are called, be sure to save the values beforehand. However, if register variables are not used on the call side, saving the saddr area for register variables is not required.

(2) Copying to base pointer (HL) of stack pointer (SP)

The stack pointer (SP) changes due to "PUSH, POP" inside functions. Therefore, the stack pointer is copied to register "HL" and used as the base pointer of arguments.

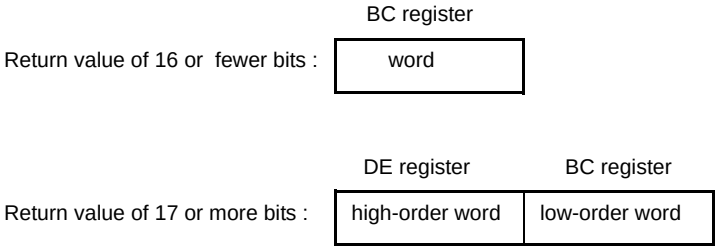
(3) Basic processing of FUNC function

After processing steps (1) and (2) are performed, the basic processing of called functions is performed.

(4) Setting the return value

If there is a return value, it is set in the "BC" and "DE" registers. If there is no return value, setting is unnecessary.

Figure 9-2. Setting Return Value

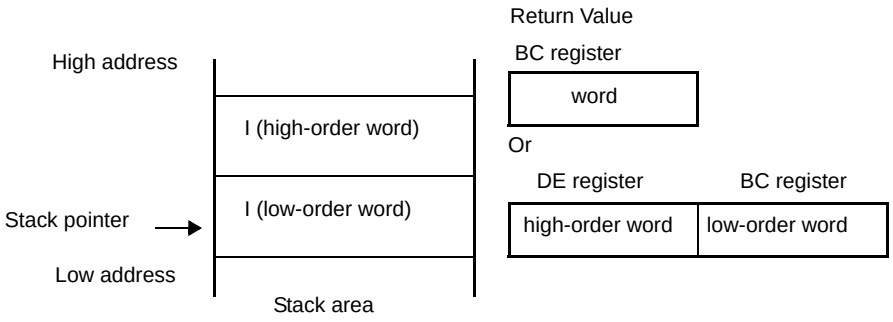


(5) Restoring the registers

Restore the saved base pointer and work register.

(6) Returning to the main function

Figure 9-3. Returning to Main Function



9.4 Calling C Language Routines from Assembly Language

This section describes the procedure for calling functions written in C language from assembly language routines.

9.4.1 Calling C language function from assembly language program

The procedure for calling a function written in the C language from an assembly language routine is as follows.

(a) Save the C work registers (AX, BC, and DE).

(b) Place the arguments on the stack.

(c) Call the C language function.

(d) Increment the value of the stack pointer (SP) by the number of bytes of arguments.

(e) Reference the return value of the C language function (in BC or DE and BC).

- This is an example of an assembly language program.

```
$PROCESSOR ( F1166A0 )

        NAME    FUNC2
        EXTRN   _CSUB
        PUBLIC  _FUNC2

@@CODE  CSEG
_FUNC2 :
        movw    ax, #20H        ; set 2nd argument ( j )
        push    ax              ;
        movw    ax, #21H        ; set 1st argument ( i )
        call    !_CSUB          ; call "CSUB ( i,  j )"
        pop     ax              ;
        ret
        END
```

(1) Saving the work registers (AX, BC, and DE)

The 3 register pairs of AX, BC, and DE are used in the C language. Their values are not restored when returning. Therefore, if the values in registers are needed, they are saved on the calling side.

Save or restore the registers before or after an argument pass code.

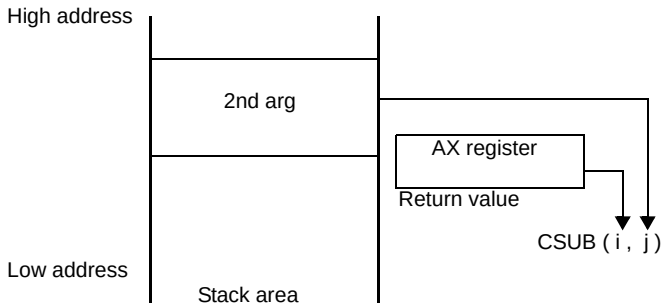
The HL register is always saved on the side of the C language when it is used in the C language.

(2) Stacking arguments

Any arguments are placed on the stack.

The following figure shows argument passing.

Figure 9-4. Argument Passing



(3) Calling a C language function

A CALL instruction calls a C language function. If the C language function is a "callt" function, the callt instruction performs the call, and if it is a "callf" function, the callf instruction performs it.

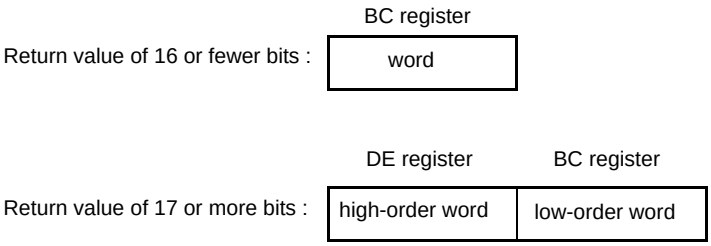
(4) Restoring the stack pointer (SP)

The stack pointer is restored by the number of bytes that hold the arguments.

(5) Referencing the return value (BC and DE)

The return value from the C language is returned as follows.

Figure 9-5. Referencing Return Values



9.5 Referencing Variables Defined in C Language

If external variables defined in a C language program are referenced in an assembly language routine, the extern declaration is used.

Underscores "_" are added to the beginning of the variables defined in the assembly language routine.

A C language program example is shown below.

```
extern  void subf ( void );

char    c = 0 ;
int     i = 0 ;

void main ( void ) {
        subf ( );
}
```

The following occurs in the assembler.

```
$PROCESSOR ( F1166A0 )

        PUBLIC  _subf
        EXTRN   _c
        EXTRN   _i

@@CODE  CSEG
_subf :
        MOV     !_c, #04H
        MOVW    AX, #07H
        MOVW    !_i, AX
        RET
        END
```

9.6 Referencing Variables Definted in Assembly Language from C Language

Variables defined in assembly language are referenced from the C language in this way.
A C language program example is shown below.

```
extern  char    c ;
extern  int     i ;

void subf ( void ) {
        c = ' A ' ;
        i = 4 ;
}
```

The following occurs in the 78K0R assembler.

```
        NAME    ASMSUB

        PUBLIC  _i
        PUBLIC  _c

ABC     DSEG    BASEP
_i :    DW      0
_c :    DB      0
        END
```

9.7 Points of Caution for Calling Between C Language Functions and Assembler Functions

- "_"(underscore)

  The 78K0R C compiler adds an underscore "_" (ASCII code "5FH") to external definitions and reference names of the object modules to be output.
  In the next C program example, "j = FUNC(i, l);" is taken as "a reference to the external name _FUNC".

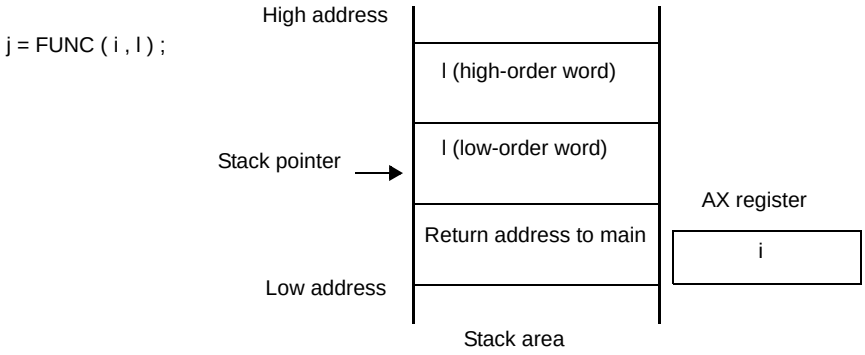
```
extern  int     FUNC ( int, long );     /* Function prototype */

void main ( void ) {
        int     i,  j ;
        long    l ;

        l = 0x54321;
        i = 1 ;
        j = FUNC ( i, l );              /* Function call */
}
```

The routine name is written as "_FUNC" in 78K0R assembler.

- Argument positions on the stack

  The arguments placed on the stack are placed from the postfix argument to the prefix argument in the direction from the High address to the Low address.

Figure 9-6. Argument Positions on Stack



CHAPTER 10 CAUTIONS

This chapter explains points of caution when coding.

(1) Kanji code classification

When using source which includes EUC code on Windows, either set the environment variable LANG78K to euc, or specify the -ze option.

(2) Include files

Functions (except declarations) defined within include files cannot be expanded in the C source.
If definitions are made within an include file, unwanted effects such as definition lines not being displayed correctly may occur at the time of debugging.

(3) When outputting assembler source

When there are assembly language descriptors such as #asm blocks or __asm statements within the C source program, the load module file creation sequence occurs in the order of: compile, assemble, link.
As in cases where there are assembly language descriptors, when assembling by outputting the assembler source first without outputting objects directly with the 78K0R C compiler, observe the following points of caution.

- If it is necessary to define symbols within #asm blocks (the area between #asm and #endasm), or within __asm statements, use symbols of eight characters or less beginning with ?L@ (e.g. ?L@01, ?L@sym, etc.). However, do not define these symbols externally (via PUBLIC declaration). Furthermore, segments cannot be defined within #asm blocks or __asm statements. If symbols beginning with the ?L@ character sequence are not used, a fatal error (F2114) will be output during assembly.
- When using variables set to extern in the C source inside of #asm blocks, if there is no reference within other parts written in C then the EXTRN is not generated and a link error occurs. Therefore, the EXTRN should be done within the #asm block when not referencing with C.
- When modifying a segment name with a #pragma section command, do not designate the segment name to be the same as the primary name of the source file name. An error (F2106) will be output during assembly.

(4) Link directive file creation

When linking objects generated by the 78K0R C compiler, if using a region other than the ROM or RAM memory of the target device, or if you want to specify any address to allocate code or data, create a link directive file and specify linker option-d when linking.
See "CHAPTER 5 LINK DIRECTIVE SPECIFICATION" or the lk.dr (under the smp folder) included with the C compiler regarding the method for creating link directive files.

Example Allocating an initial value null external variable (other than an sreg variable) to external memory in a C source file.

(a) Modify the section name used for initial value null external variable at the head of the C source.

```
#pragma section @@DATAL EXTDATA
    :
```

Caution To initialize or ROMize the modified segment, do so by modifying the startup routine.

(b) Create the lk78k0r.dr link directive file.

```
memory  EXTRAM  : ( 040000H, 1000H )
merge   EXTDATA : = EXTRAM
```

Pay attention to the following points when creating link directive files.

- When linking, if using stack resolution symbol creation designation option -s, it is recommended to secure the stack region with the link directive file's memory directive, and explicitly define the name of the secured stack region. If the region name is excluded, RAM memory (except for SFR memory) will be used as the stack region.

Example When the link directive file lk78k0r.dr has been added to.

```
memory  EXTRAM  : ( 040000H, 1000H )
memory  STK     : ( 0FB000H, 100H )
merge   EXTDATA : = EXTRAM
```

The command line is as follows.

```
C>lk78k0r s0rml.rel prime.rel -bcl0rm.lib -sSTK -dlk78k0r.dr
```

- When linking to the defined memory region, the following link error may be output.

```
RA78K0R error E3206 : Segment 'xxx' can't allocate to memory-ignored.
```

This means that the specified segment cannot be allocated due to insufficient space in the specified memory region.
The method for dealing with this is divided broadly into the following 3 steps.

<1> Investigate the size of the segment that cannot be allocated (see the .map file).
However, depending on the type of segment specified by the error, the method for investigating the segment size differs as follows.

- For segments automatically generated at time of compiling
  Investigate the segment size in the map file created from linking.
- For user-created segments
  Investigate the size of the segment which was not allocated in the assembly list file (.prn).

<2> Based on the segment size found above, increase the memory size where the segment will be allocated with the directive file.

<3> Specify option -d for the directive file specification and link.

(5) When using the va_start macro

Because the offset of the first argument differs depending on the function, operation of the va_start macro defined in stdarg.h is not guaranteed.
Use the macro accordingly as follows.

- When specifying the first argument, use the va_starttop macro.

(6) Regarding the startup routines and libraries

- Use the same startup routine and library version offered as the version of the execute form file (cc78k0r.exe) being used.
- In floating-point compatible sprintf, vprintf and vsprintf for the, values below the accuracy level of the "%f", "%e", "%E", "%g" and "%G" specified conversion results are rounded down. Additionally, "%f" conversion occurs even if the "%g" and "%G" specified conversion results are above the accuracy level.
  In floating-point compatible sscanf and scanf, if no valid character is read at "%f", "%e", "%E", "%g" and "%G" specification then the conversion result will be +0, and if "±" is the only character recognized then the conversion result will be ± 0.

(7) When ROMization is performed

ROMization is the process of allocating an initial value such as an external variable with an initial value to ROM then copying it to RAM when the system is executed. The 78K0R C compiler by default generates code to be ROMized. Therefore it is necessary to link to a start-up routine that includes ROMization processing when linking.

ROMization processing of far memory is not included in the startup routines for small and medium models. When a variable has been allocated to far memory using the __far modifier, use a large model startup routine.

The CA78K0R offers the following startup routines, all of which include ROMization processing.

If using flash memory self overwrite mode, please see "(3) Uses of startup routines".

When not using C standard library memory	s0rm.rel, s0rl.rel
When using C standard library memory	s0rml.rel, s0rll.rel

A usage example is shown below.

The -s option is a stack symbol (_@STBEG, _@STEND) automatically generated option.

```
C>lk78k0r s0rl.rel sample.rel -s -bcl0rxm.lib -bcl0rm.lib -osample.lmf
```

sample.rel	User programmed object module file
s0rl.rel	Startup routine
cl0rxm.lib	Multiplier library
cl0rdm.lib	Multiplier/divider library
cl0rm.lib	Runtime library, standard library

Cautions 1. Be sure to link the startup routine first.

2. When creating user-generated libraries, separate them from the libraries provided by CA78K0R, and specify them ahead of the CA78K0R libraries when linking.

3. Do not add user functions to the CA78K0R libraries

4. When a floating-point library (cl0r*f.lib) is used, it is also necessary to link to the normal library (cl0r*.lib).

Example When using floating-point compatible sprintf, sscanf, printf, scanf, vprintf and vsprintf.

```
-bmylib.lib -bcl0rmf.lib -bcl0rm.lib
```

Example When using sprintf, sscanf, printf, scanf, vprintf and vsprintf not compatible with floating point.

```
-bmylib.lib -bcl0rm.lib -bcl0rmf.lib
```

(8) Prototype declaration

In a function prototype declaration, if the function type is not specified, an error (E0301, E0701) results.

```
f ( void );     /* E0301 : Syntax error */
                /* E0701 : External definition syntax */
```

In such a case, specify the function type.

```
int     f ( void );
```

(9) Error message output

Outside the function, if there is a spelling error in the keyword at the beginning of a line, the display position of the error line may be shifted or an improper error may be output.

```
extren int      i ;             /* extern is misspelled.  An error does not occur here.*/
/* comment */
void f ( void );
[EOF]                           /* Error such as E0712 */
```

(10) Comment input in the preprocessor directive

If a comment is inserted at the beginning or in the middle of a preprocessor directive in the function form macro row, an error occurs (E0803, E0814, E0821, etc.).

```
/* com1 */      #pragma         sfr                     /* E0803 */
/* com2 */      #define         ONE     1               /* E0803 */
#define         /* com3 */      TWO     2               /* E0814 */
#ifdef          /* com4 */      ANSI_C                  /* E0814 */

/* com5 */      #endif
#define         SUB ( p1, /* com6 */ p2 ) p2 = p1       /* E0821 */
```

In such a case, insert the comment after the preprocessor directive.

```
#pragma         sfr                                     /* com1 */
#define         ONE     1                               /* com2 */
#define         TWO     2                               /* com3 */
#ifdef          ANSI_C                                  /* com4 */

#endif                                                  /* com5 */
#define         SUB ( p1, p2 ) p2 = p1                  /* com6 */
```

(11) Tag usage for structure, union and enum

In the function prototype declaration, if (structure, union and enum) tags are used before they are defined, a warning occurs if conditions (a) below are met, and an error occurs if conditions (b) below are met.

(a) If the tag is used to define the pointers to structure and union in the argument declaration, a warning (W0510) occurs when the function is called.

```
void func ( int, struct st );

struct  st {
        char    memb1 ;
        char    memb2 ;
} st[ ] = {
        { 1, ' a ' }, { 2, ' b ' }
};

void caller ( void ) {
                        /* W0510 Pointer mismatch */
        func ( sizeof ( st ) / sizeof ( st[0] ), st );
}
```

(b) If the tag is used to designate structure, union and enum types for the return value declaration and argument statement, an error (E0737) will occur.

```
                        /* E0737 Undeclared structure/union/enum tag */
void func1 ( int, struct st );
                        /* E0737 Undeclared structure/union/enum tag */
struct  st      func2 ( int );
struct  st {
        char    memb1 ;
        char    memb2 ;
};
```

In such a case, first define the structure, union and enum tags.

(12) Initializing arrays, structures and unions within a function

Within a function, array, structure and union initialization cannot be performed using other than a static variable address, constant, and character string

```
void f ( void );

void f ( void ) {
        char    *p, *p1, *p2 ;
        char    *ca[3] = { p, p1, p2 };    /* Error( E0750 ) */
}
```

In such a case, enter an assignment statement and substitute it in place of initialization.

```
void f ( void );

void f ( void ) {
        char    *ca[3] ;
        char    *p, *p1, *p2 ;
        ca[0] = p ;  ca[1] = p1 ; ca[2] = p2 ;
}
```

(13) extern callt functions

If an extern callt function address is referenced in a function table initialization and the callt function is called in the same module, the assemble list becomes illegal and an error occurs during assembly.

```
callt   extern  void funca ( void );
callt   extern  void funcb ( void );
callt   extern  void funcc ( void );

static  void ( * const func[] ) ( ) = {
        funca, funcb, funcc
};

void func2 ( void ) {
        funcc ( );
        funcb ( );
        funca ( );
}
```

In such a case, separate the function table from the function call module.

(14) Structure-returning functions

sWhen a function returns the structure itself, if an interrupt occurs during return processing of the return value and during interrupt processing the same function is called, the return value after completion of interrupt processing will be invalid.

```
struct  str {
        char    c ;
        int     i ;
        long    l ;
} st ;

struct  str     func ( ) {
        /* Interrupt occurrence */
          :
}

void main ( ) {
        st = func ( );  /* Interrupt occurrence */
}
```

During the processing excerpt above, if the func function is called at the interrupt destination, st may break down.

(15) Union initialization

In initialization of the union, which holds structure, union and array as members, if the initializer list is designated via nesting, an error (E0750) occurs.

```
struct  Ss {
        int     d1, d2 ;
};
union   Au {
        struct  Ss t1 ;
} u = { { 1, 2 } };    /* E0750  Initializer syntax */
```

In such a case, do not use nesting to designate the union initializer list.

```
struct  Ss {
        int     d1, d2 ;
};
union   Au {
        struct  Ss t1 ;
} u = { 1, 2 };
```

(16) Memory initialization directives

If memory initialization directives DB, DW and DG are input with the data segment (DSEG), the object code will be output, but a warning (W4301) will occur at the object converter. This is because code exists at an address other than the ROM region (the coding region).

If ROM code is called (operations called across processing and tape-out) in this state, an error occurs.

(17) Memory directives

The default memory region name of each device cannot be erased.

Set the memory size for default memory names not being used to 0.

However, some segments may be assigned to the default regions, so be careful when modifying region names.

See the user manual of each device regarding default memory region names.

(18) Segment names

When inputting a segment name, do not give a segment a name that is the same as the primary name of the source file name. Abort error F2106 will occur at assembly.

(19) EQU definition of SFR name

An SFR name can be designated in the operand of an EQU directive, but if the name of an SFR outside the saddr area is designated an assemble error will occur.

(20) Specifying section start addresses

The size of a section that specifies a start address with a #pragma section directive is always an even number.

APPENDIX A INDEX

Symbols

#asm - #endasm ... 91
#pragma bcd ... 133
#pragma BRK ... 106
#pragma DI ... 103
#pragma directive ... 74
#pragma div ... 131
#pragma EI ... 103
#pragma ext_func ... 153
#pragma ext_table ... 147
#pragma HALT ... 106
#pragma inline ... 162
#pragma interrupt ... 95
#pragma mul ... 129
#pragma name ... 126
#pragma NOP ... 106
#pragma opc ... 136
#pragma rot ... 127
#pragma rtos_interrupt ... 138
#pragma rtos_task ... 142
#pragma section ... 114
#pragma sfr ... 86
#pragma STOP ... 106
#pragma vect ... 95
?A0nnnnn ... 204
?BSEG ... 204
?CSEG ... 204
?CSEGB ... 204
?CSEGFX ... 204
?CSEGMIP ... 204
?CSEGOB0 ... 204
?CSEGP64 ... 204
?CSEGSI ... 204
?CSEGT0 ... 204
?CSEGU64 ... 204
?CSEGUP ... 204
?DSEG ... 204
?DSEGBP ... 204
?DSEGP64 ... 204
?DSEGS ... 204
?DSEGSP ... 204
?DSEGU64 ... 204
?DSEGUP ... 204
_@BRKADR ... 773
_@DIVR ... 774
_@FNCENT ... 774
_@FNCTBL ... 774
_@LDIVR ... 774
_@MEMBTM ... 773
_@MEMTOP ... 773
_@SEED ... 774
_@STBEG ... 767, 768
_@TOKPTR ... 774

A

abort ... 649
abs ... 652
absolute address allocation specification ... 77, 164
absolute assembler ... 19
absolute segment ... 195, 263
absolute term ... 248
acos ... 697
acosf ... 720
ADDRESS ... 204
ADDRESS term ... 250
aggregate type ... 70
allocating ROM data specification ... 77
alphabetic characters ... 201
alphanumeric characters ... 201
AND operator ... 223
ANSI ... 72
area reservation directives ... 286
arithmetic operator ... 213
array type ... 69
asin ... 698
asinf ... 721

__asm ... 91
ASM statement ... 75, 91
assemble target type specification control instruction ... 328
assemble termination directive ... 325
assembler options ... 327
assembly language ... 18
assembly list control instructions ... 344
assert ... 592
assert.h ... 748
__assertfail ... 743
AT ... 570
AT relocation attribute ... 266, 270, 274
atan ... 699
atan2 ... 700
atan2f ... 723
atanf ... 722
atexit ... 650
atof ... 658
atoi ... 641
atol ... 642

B

backward reference ... 260
BASE relocation attribute ... 266
BASEP relocation attribute ... 270
BCD operation function ... 76, 133
binary ... 206
binary constant ... 76, 124
BIT ... 204
bit field ... 108
bit field declaration ... 75, 76, 108, 110
bit segment ... 195, 263
bit symbol ... 253
bit type variable ... 75, 88
BITPOS operator ... 244
__boolean ... 88
boolean type variable ... 75, 88
BR directive ... 306
branch instruction automatic selection directives ... 305
BRK ... 106
brk ... 656
bsearch ... 665
BSEG directive ... 273
byte separation operator ... 236

C

C language ... 18
CALL directive ... 308
calloc ... 645
callt function ... 75, 78
callt/__callt ... 78
CALLT0 relocation attribute ... 266
ceil ... 715
ceilf ... 738
changing compiler output section name ... 76
char type ... 66
character set ... 200
character string constant ... 206
character type ... 69
CLRB ... 459
CLRW ... 467
code segment ... 195, 263
comment field ... 209, 398
compiler output section name is changed ... 114
COMPLETE ... 570
concatenation ... 397
COND control instruction ... 354
conditional assembly control instructions ... 366
constant ... 206
control instructions ... 327
cos ... 701
cosf ... 724
cosh ... 704
coshf ... 727
CPU control instruction ... 75, 106
cross-reference list output specification control instructions ... 335
CSEG directive ... 265
cstart*.asm ... 765
cstart.asm ... 763, 765
cstartn.asm ... 763, 765
ctype.h ... 586, 744

D

data insertion function ... 76, 136
data segment ... 195, 263

DATAPOS operator ... 243
DB directive ... 287
DBIT directive ... 295
DEBUG control instruction ... 331
debug information output control instructions ... 330
DEBUGA control instruction ... 333
decimal ... 206
DG directive ... 291
DGL control instruction ... 391
DGS control instruction ... 391
DI ... 103
directive file ... 566
directives ... 262
__directmap ... 164
div ... 654
division function ... 76, 131
DS directive ... 293
DSEG directive ... 269
DW directive ... 289

E

EI ... 103
EJECT control instruction ... 345
ELSE control instruction ... 378
_ELSEIF control instruction ... 376
ELSEIF control instruction ... 373
END directive ... 326
ENDIF control instruction ... 381
ENDM directive ... 323
enumeration type ... 66
EQ operator ... 227
EQU directive ... 280
_errno ... 773
errno.h ... 588
error.h ... 587
EUC ... 93
exit ... 651
EXITM directive ... 320
exp ... 707
expf ... 730
EXTBIT directive ... 299
external reference term ... 248
EXTRN directive ... 297
ext_tsk ... 142

F

fabs ... 716
fabsf ... 739
FIXED relocation attribute ... 266
flash area allocation method ... 76, 146
flash area branch table and flash area allocation ... 76, 147
float.h ... 590
floating point type ... 67
floor ... 717
floorf ... 740
fmod ... 718
fmodf ... 741
FORMFEED control instruction ... 361
forward reference ... 260
free ... 646
frexp ... 708
frexpf ... 731
function of function call from boot area to flash area ... 76, 153
function type ... 70

G

GE operator ... 230
GEN control instruction ... 350
general register ... 207
general register pairs ... 207
getchar ... 635
gets ... 636
global symbol ... 395
GT operator ... 229

H

HALT ... 106
hardware initialization function ... 768
hdwinit function ... 765, 768
header File ... 586
hexadecimal ... 206
HIGH operator ... 237
HIGHW operators ... 240
how to use the saddr area ... 75
how to use the sfr area ... 75

I

_IF control instruction ... 371
IF control instruction ... 367
INCLUDE control instruction ... 341
include control instruction ... 340
incomplete type ... 69
integer type ... 66
__interrupt ... 101
interrupt function ... 75, 95, 103
interrupt function qualifier ... 75, 101
Interrupt handler for RTOS ... 138
interrupt handler for RTOS ... 76
Interrupt handler qualifier for RTOS ... 141
interrupt handler qualifier for RTOS ... 76
__interrupt_brk ... 101
IRP directive ... 318
IRP-ENDM block ... 318
isalnum ... 599
isalpha ... 595
isascii ... 606
iscntrl ... 605
isgraph ... 604
islower ... 597
isprint ... 603
ispunct ... 602
isspace ... 601
isupper ... 596
isxdigit ... 600
itoa ... 660
IXRAM relocation attribute ... 266

K

Kanji (2-byte character) ... 75, 93
Kanji code control instruction ... 389
KANJICODE control instruction ... 390

L

label ... 202
labs ... 653
LANG78K ... 93
ldexp ... 709
ldexpf ... 732
ldiv ... 655
LE operator ... 232
LENGTH control instruction ... 364
limits.h ... 588
link directive ... 566, 572, 767
linkage directives ... 296
LIST control instruction ... 347
LOCAL directive ... 313
local symbol ... 395
log ... 710
log10 ... 711
log10f ... 734
logf ... 733
logic operator ... 221
longjmp ... 616
LOW operator ... 238
LOWW operators ... 241
LT operator ... 231
ltoa ... 661

M

macro ... 392
macro definition ... 392
MACRO directive ... 311
macro directives ... 310
macro expansion ... 394
macro name ... 202
macro operator ... 397
malloc ... 647
MASK operator ... 245
math.h ... 589, 747
matherr ... 719
memchr ... 682
memcmp ... 679
memcpy ... 673
memmove ... 674
MEMORY ... 570
memory directive ... 567
memory initialization directives ... 286
memory manipulation function ... 76, 162
memory model ... 70
Memory model specification ... 173
memory model specification ... 77
memory space ... 71
memset ... 690

MERGE ... 570
method of int expansion limitation of argument/return value ... 76, 159
mirror source area specification ... 76
MIRRORP relocation attribute ... 266
mkstup.bat ... 761, 763
mnemonic field ... 205, 398
MOD operator ... 218
modf ... 712
modff ... 735
modular programming ... 19
module header ... 194
module name ... 202
module name changing function ... 76, 126
module tail ... 195
MOV ... 452
MOVS ... 460
MOVW ... 462
multiplication function ... 76, 129

N

name ... 202
NAME directive ... 304
NE operator ... 228
near/far area specification ... 77, 168
NOCOND control instruction ... 355
NODEBUG control instruction ... 332
NODEBUGA control instruction ... 334
NOFORMFEED control instruction ... 362
NOGEN control instruction ... 352
NOLIST control instruction ... 348
NONE ... 93
NOP ... 106
NOSYMLIST control instruction ... 339
NOT operator ... 222
NOXREF control instruction ... 337
NUMBER ... 204
NUMBER term ... 250
numeric constant ... 206

O

object module name declaration directive ... 303
octal ... 206
ONEW ... 466
__OPC ... 136
operand ... 254, 259
operand field ... 205, 398
operator ... 210
OPT_BYTE relocation attribute ... 266
optimized branching directive ... 27
OR operator ... 224
ORG directive ... 277
other operator ... 246

P

PAGE64KP relocation attribute ... 266, 270
pow ... 713
powf ... 736
printf ... 631
PROCESSOR control instruction ... 329
PUBLIC directive ... 301
__putc ... 639
putchar ... 637
puts ... 638

Q

-ql option ... 78
qsort ... 666

R

rand ... 663
realloc ... 648
re-entrant ... 592
referencing macro ... 393
register ... 80
register bank ... 70
register bank is specified ... 95
register variable ... 75, 80
REGULAR ... 568, 570
relocatable assembler ... 19
relocatable term ... 248
relocation attribute ... 248, 266, 270, 274
repgetc.bat ... 761
repputc.bat ... 761
repputcs.bat ... 761
reprom.bat ... 761
repselo.bat ... 761
repselon.bat ... 761

REPT directive ... 316
REPT-ENDM block ... 316
repvect.bat ... 761
RESET control instruction ... 387
reset vector ... 768
rolb ... 127
rolw ... 127
rom.asm ... 765
ROMization ... 760, 775
ROMization processing ... 768, 772
ROMization routine ... 761
rorb ... 127
rorw ... 127
rotate function ... 76, 127
RTOS ... 72
__rtos_interrupt ... 141
runtime library ... 775

S

s0r*.rel ... 765
SADDR relocation attribute ... 270
SADDRP relocation attribute ... 270
sbrk ... 657
scanf ... 632
section name related to ROMization ... 120
SECUR_ID relocation attribute ... 266
segment ... 195
segment definition directive ... 263
segment location directive ... 569
SEQUENT ... 570
SET control instruction ... 385
SET directive ... 284
setjmp ... 615
setjmp.h ... 586, 744
sfr area ... 86
sfr variable ... 86
shift operator ... 233
SHL operator ... 235
SHR operator ... 234
signed integer type ... 66
sin ... 702
sinf ... 725
sinh ... 705
sinhf ... 728
SJIS ... 93
source module ... 194
special characters ... 201, 207
special function register ... 207
special operator ... 242
sprintf ... 623
sqrt ... 714
sqrtf ... 737
srand ... 664
sreg ... 82
sscanf ... 627
stack change specification ... 96
stack pointer ... 768
standard library ... 775
startup ... 760
startup module ... 772
startup routine ... 120, 756, 784
statement ... 200
stdarg.h ... 586, 745
stddef.h ... 589
stdio.h ... 586, 745
stdlib.h ... 587, 745
STOP ... 106
strbrk ... 667
strcat ... 677
strchr ... 683
strcmp ... 680
strcoll ... 693
strcpy ... 675
strcspn ... 686
strerror ... 691
string.h ... 587, 746
stritoa ... 669
strlen ... 692
strltoa ... 670
strncat ... 678
strncmp ... 681
strncpy ... 676
strpbrk ... 687
strrchr ... 684
strsbrk ... 668
strspn ... 685

strstr ... 688
strtod ... 659
strtok ... 689
strtol ... 643
strtoul ... 644
structure type ... 70
strultoa ... 671
strxfrm ... 694
subroutine ... 392
SUBTITLE control instruction ... 358
symbol ... 395
symbol attribute ... 204
symbol definition directives ... 279
symbol field ... 398
SYMLIST control instruction ... 338

T

TAB control instruction ... 365
tan ... 703
tanf ... 726
tanh ... 706
tanhf ... 729
task ... 142
Task function for RTOS ... 142
task function for RTOS ... 76
TITLE control instruction ... 356
toascii ... 609
TOL_INF control instruction ... 391
tolow ... 613
_tolower ... 612
tolower ... 608
toup ... 611
toupper ... 607

U

ultoa ... 662
union type ... 70
UNIT relocation attribute ... 266, 270, 274
UNIT64KP relocation attribute ... 266, 270
UNITP relocation attribute ... 266, 270
unsigned integer type ... 66
usage with saddr automatic allocation option of external variables/external static variables ... 75
usage with saddr automatic allocation option of internal static variables ... 75
use the saddr area ... 82

V

va_arg ... 620
va_end ... 621
va_start ... 618
va_starttop ... 619
vprintf ... 633
vsprintf ... 634

W

WIDTH control instruction ... 363
word separation operators ... 239

X

XCH ... 456
XCHW ... 465
XOR operator ... 225
XREF control instruction ... 336

Z

-zb ... 159
-zf ... 146

For further information,
please contact:

NEC Electronics Corporation
1753, Shimonumabe, Nakahara-ku,
Kawasaki, Kanagawa 211-8668,
Japan
Tel: 044-435-5111
http://www.necel.com/

[America]

NEC Electronics America, Inc.
2880 Scott Blvd.
Santa Clara, CA 95050-2554, U.S.A.
Tel: 408-588-6000
800-366-9782
http://www.am.necel.com/

[Europe]

NEC Electronics (Europe) GmbH
Arcadiastrasse 10
40472 Düsseldorf, Germany
Tel: 0211-65030
http://www.eu.necel.com/

Hanover Office
Podbielskistrasse 166 B
30177 Hannover
Tel: 0 511 33 40 2-0

Munich Office
Werner-Eckert-Strasse 9
81829 München
Tel: 0 89 92 10 03-0

Stuttgart Office
Industriestrasse 3
70565 Stuttgart
Tel: 0 711 99 01 0-0

United Kingdom Branch
Cygnus House, Sunrise Parkway
Linford Wood, Milton Keynes
MK14 6NP, U.K.
Tel: 01908-691-133

Succursale Française
9, rue Paul Dautier, B.P. 52
78142 Velizy-Villacoublay Cédex
France
Tel: 01-3067-5800

Sucursal en España
Juan Esplandiu, 15
28007 Madrid, Spain
Tel: 091-504-2787

Tyskland Filial
Täby Centrum
Entrance S (7th floor)
18322 Täby, Sweden
Tel: 08 638 72 00

Filiale Italiana
Via Fabio Filzi, 25/A
20124 Milano, Italy
Tel: 02-667541

Branch The Netherlands
Steijgerweg 6
5616 HS Eindhoven
The Netherlands
Tel: 040 265 40 10

[Asia & Oceania]

NEC Electronics (China) Co., Ltd
7th Floor, Quantum Plaza, No. 27 ZhiChunLu Haidian
District, Beijing 100083, P.R.China
Tel: 010-8235-1155
http://www.cn.necel.com/

Shanghai Branch
Room 2509-2510, Bank of China Tower,
200 Yincheng Road Central,
Pudong New Area, Shanghai, P.R.China P.C:200120
Tel:021-5888-5400
http://www.cn.necel.com/

Shenzhen Branch
Unit 01, 39/F, Excellence Times Square Building,
No. 4068 Yi Tian Road, Futian District, Shenzhen,
P.R.China P.C:518048
Tel:0755-8282-9800
http://www.cn.necel.com/

NEC Electronics Hong Kong Ltd.
Unit 1601-1613, 16/F., Tower 2, Grand Century Place,
193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: 2886-9318
http://www.hk.necel.com/

NEC Electronics Taiwan Ltd.
7F, No. 363 Fu Shing North Road
Taipei, Taiwan, R. O. C.
Tel: 02-8175-9600
http://www.tw.necel.com/

NEC Electronics Singapore Pte. Ltd.
238A Thomson Road,
#12-08 Novena Square,
Singapore 307684
Tel: 6253-8311
http://www.sg.necel.com/

NEC Electronics Korea Ltd.
11F., Samik Lavied'or Bldg., 720-2,
Yeoksam-Dong, Kangnam-Ku,
Seoul, 135-080, Korea
Tel: 02-558-3737
http://www.kr.necel.com/

G0706